Honeywell

PL/I REFERENCE MANUAL

SERIES 60 (LEVEL 66)/6000

SOFTWARE

Honeywell

PL/I REFERENCE MANUAL

SERIES 60 (LEVEL 66)/6000

SUBJECT:

Reference Manual for PL/I in a GCOS Environment.

SOFTWARE SUPPORTED:

Series 60 (Level 66)/6000 Software Release PL 1.0

DATE:

March 1976

ORDER NUMBER:

DE05, Rev. 0

PREFACE

This manual is a combined tutorial and reference manual for GCOS PL/I. This manual is intended for a programmer who is experienced in the use of high-level languages, such as FORTRAN, COBOL, or ALGOL.

The sections of this manual are organized to reflect the main features of the language. After the introductory section, three aspects of data values are considered: values as abstractions, values in storage, and the conversion of values from one storage type to another. Next, the overall syntax of a program is considered, ranging from small constructs (such as the identifier) to intermediate constructs (the statements) to large constructs (the blocks). Against this background, the declaration of identifiers and the management of storage are described. Then, the features that are used to compute and store values are considered, followed by a description of the features that are used to determine the sequence in which program statements are executed. Finally, the statements for input/output are given. The manual has an appendix that gives the syntax of all of the statements of PL/I.

Two related Honeywell publications on GCOS PL/I are cited in the Introduction of this manual. They are the PL/I Language Manual (AG94) and the GCOS PL/I User's Guide (DE04).

© 1976, Honeywell Information System Inc.

File No: IP33,1733

DE05

CONTENTS

		Page
Section I	Introduction.	1-1
	The Main Features of PL/I.	1-1
	Data Description.	1-2
	Program Structure	1-3
	Computation	1-4
	Sequencing.	1-4
	Input/Output.	1-5
	Applications of PL/I	1-5
	Scientific Programming.	1-5
	Business Programming.	1-7
	System Programming.	1-8
	Data	1-8
	Program Structure.	1-8
	Efficiency	1-9
	Program Validity	1-9
	Examples of Invalid Programs.	1-9
	The Interpretation of Invalid Programs.	1-11
	Suggestions for the Study of PL/I.	1-12
	Introductory Texts.	1-12
	The PL/I Reference Manual	1-12
	The Role of Examples in This Manual.	1-13
	The Role of Guidelines in This Manual. . . .	1-13
	The Organization of This Manual.	1-13
	Related Manuals	1-14
Section II	Values.	2-1
	Arithmetic Values.	2-1
	String Values.	2-2
	Address Values	2-3
	Statement Values.	2-3
	Locator Values.	2-4
	File Values	2-4
	Area Values.	2-4
	Aggregate Values	2-4
	The Classification of Values	2-5
Section III	Value Storage	3-1
	Storage Units.	3-1
	Storage Types.	3-2
	Arithmetic Storage	3-4
	Arithmetic Data Types	3-5
	The Mode Attribute	3-5
	The Scale Attribute.	3-5
	The Base Attribute	3-6
	The Precision Attribute.	3-7
	Abbreviations and Defaults	3-8
	Examples of Arithmetic Storage Units.	3-10
	FIXED DECIMAL Storage Units.	3-11
	FIXED BINARY Storage Units	3-12
	FLOAT BINARY Storage Units	3-13
	FLOAT DECIMAL Storage Units.	3-13
	COMPLEX Storage Units.	3-14
	Guidelines for Arithmetic Data Types.	3-14

	Page
The Choice of Mode	3-14
The Choice of Scale and Base	3-15
The Choice of Precision	3-15
Ordinary String Storage	3-16
Ordinary String Data Types	3-16
The String-Type Attribute	3-16
The Variability Attribute	3-17
Abbreviations and Defaults	3-17
Examples of Ordinary String Storage	3-18
Character Storage Units	3-18
Bit Storage Units	3-19
Guidelines for Ordinary String Data Types	3-19
Pictured String Storage	3-20
Pictured Data Types	3-20
The Picture Attribute	3-21
The Classification of Indicators	3-21
The Classification of Pictures	3-22
The Mode Attribute	3-23
Abbreviations and Defaults	3-23
Interpreting Pictured Storage	3-23
The Character-String Interpretation of Pictured Storage	3-24
Related Character Types	3-24
Character-String Assignments	3-25
Character-String Fetches	3-26
The Arithmetic Interpretation of Pictured Storage	3-26
Related Arithmetic Data Types	3-26
Arithmetic Assignments	3-27
Arithmetic Fetches	3-28
Fixed-Point Pictures	3-29
The No-Suppression Digit Indicator	3-30
The Decimal-Point Indicator	3-32
The Sign Indicators	3-33
The Dollar Indicator	3-34
The Zero-Suppression Digit Indicators	3-35
The Drifting-Sign Digit Indicators	3-36
The Drifting-Dollar Digit Indicator	3-37
The Insertion-Character Indicators	3-38
The Arithmetic Decimal-Point Indicator	3-39
The Scale-Factor Indicator	3-40
Floating-Point Pictures	3-41
The Floating-Point Indicators	3-41
The Scale-Factor Indicator	3-42
Complex Pictures	3-42
Non-Numeric Pictures	3-43
The Non-Numeric Indicators	3-43
Guidelines for Pictured Storage	3-44
Address Storage	3-44
Address Attributes	3-45
An Abbreviation	3-46
An Example of Address Storage	3-46
Area Storage	3-46
The AREA Attribute	3-47
A Default Rule	3-48
An Example of Area Storage	3-48
Aggregate Storage	3-49
Structures	3-49
Level Numbers	3-49
Structure Storage Types	3-50
Examples	3-51
Arrays	3-53

		Page
	The DIMENSION Attribute.	3-54
	Abbreviations and Defaults	3-55
	Array Storage Types.	3-56
	Examples	3-57
	Guidelines for Aggregates	3-60
	Alignment.	3-61
	The Alignment Attribute	3-62
	Abbreviations and Defaults.	3-63
Section IV	Value Conversion.	4-1
	Contexts That Force Conversion	4-1
	General Contexts.	4-2
	Assignment Statements.	4-2
	Assignment-Like Constructs	4-3
	Arguments and Results.	4-3
	Built-In Functions and Expressions	4-4
	Special Contexts.	4-4
	Integer Targets.	4-5
	Character-String Targets	4-5
	Bit-String Targets	4-6
	Locator Targets.	4-6
	Data Type Conversion	4-7
	Arithmetic Targets.	4-7
	Examples of Arithmetic to Arithmetic Conversion.	4-9
	Examples of String to Arithmetic Conversion.	4-10
	Character-String Targets.	4-10
	Examples of Character-String to Character-String Conversion	4-12
	Examples of Bit-String to Character-String Conversion.	4-12
	Examples of Arithmetic to Character-String Conversion.	4-13
	FLOAT DECIMAL Values.	4-13
	FIXED DECIMAL Values.	4-13
	FLOAT BINARY Values	4-14
	FIXED BINARY Values	4-15
	COMPLEX Values.	4-15
	Bit-String Targets.	4-16
	Examples of Bit-String to Bit-String Conversion.	4-16
	Examples of Character-String to Bit-String Conversion.	4-17
	Examples of Arithmetic to Bit-String Conversion.	4-17
	FIXED BINARY Values	4-17
	FIXED DECIMAL Values.	4-17
	FLOAT Values.	4-18
	COMPLEX Values.	4-18
	Locator Targets	4-18
	Aggregate Type Conversion.	4-18
	An Example of Aggregate Conversion.	4-19
	Conditions for Conversions	4-20
	The SIZE and FIXEDOVERFLOW Conditions	4-20
	The OVERFLOW and UNDERFLOW Conditions	4-21
	The CONVERSION Condition.	4-21
	The STRINGSIZE Condition.	4-22
	Guidelines for Conversion Conditions.	4-22
Section V	Program Syntax.	5-1
	Characters	5-2

		Page
	Lexemes	5-2
	Identifiers	5-3
	Keyword vs. Name	5-3
	Guidelines for Identifiers	5-4
	Literal Constants	5-5
	Punctuators	5-6
	Operators	5-6
	Pictures	5-7
	Isubs	5-7
	The %Include Macro	5-7
	Macro Interpretation	5-8
	An Example Macro	5-8
	Guidelines for Macros	5-9
	Separators	5-10
	The Separation Rules	5-10
	The Classification of Lexemes	5-12
	Statements	5-13
	The Statement Prefix	5-13
	Condition Prefixes	5-13
	Label Prefixes	5-13
	The Statement Body	5-14
	The Classification of Statements	5-16
	Program Structures	5-17
	Groups	5-17
	Blocks	5-18
	Summary of the Program Structures	5-19
	External Procedures and the Program	5-20
Section VI	Declarations	6-1
	The Establishment of Declarations	6-1
	A Preliminary Example of the Establishment of Declarations	6-2
	Containment and Immediate Containment	6-3
	The DECLARE Statement	6-4
	Simple Declarations	6-4
	Structure Declarations	6-5
	Short Forms of Declarations	6-6
	Abbreviations and Defaults	6-6
	Combining Declarations	6-6
	Factoring Declarations	6-6
	Guidelines for DECLARE Statements	6-7
	Label Prefixes	6-8
	Label Constant Names	6-8
	Entry Constant Names	6-8
	Format Constant Names	6-9
	Contextual and Implicit Declarations	6-10
	An Example of Contextual and Implicit Declarations	6-10
	Guidelines for Contextual and Implicit Declarations	6-11
	Special Facilities for Declaration	6-11
	The LIKE Attribute	6-11
	The Form of the LIKE Attribute	6-12
	The Interpretation of the LIKE Attribute	6-12
	Examples of the LIKE Attribute	6-13
	Guidelines for the LIKE Attribute	6-13
	The DEFAULT Statement	6-13
	Examples of the DEFAULT Statement	6-14
	Guidelines for the DEFAULT Statement	6-15
	The Resolution of Name References	6-15

		Page
	A Preliminary Example of the Resolution of Name References	6-16
	The Name-Sequence Set for a Declaration	6-16
	The Name-Sequence for a Name Reference	6-17
	The Applicability of Declarations	6-17
	The Resolution Rules	6-18
	Attributes	6-20
	Complete Attribute Sets	6-20
	Variable Names	6-21
	Constant Names	6-22
	Built-In Function Names	6-23
	Condition Names	6-23
	Generic Names	6-23
	The Classification of Attributes	6-24
Section VII	Storage Management	7-1
	Preliminary Examples of Storage Management	7-2
	Fundamentals of Storage Management	7-3
	Storage Regions	7-3
	Region Diagrams	7-4
	Storage Management Operations	7-5
	A Large Example of Storage Management	7-6
	The Management Class	7-10
	The Form of the Management Class	7-10
	Abbreviations and Defaults	7-11
	Remarks on the Default Rules	7-12
	Usage Categories	7-12
	Variables	7-13
	Constants	7-13
	Scopes	7-13
	Internal Names	7-13
	External Names	7-14
	Guidelines for the Scope	7-14
	The Correspondence Between Names and Storage	7-14
	Storage Classes	7-15
	Automatic Variables	7-15
	Variable Expressions in Attributes for Automatic Variables	7-16
	The Saving of Variable Extents for Automatic Variables	7-16
	Static Variables	7-17
	Variable Expressions in Attributes for Static Variables	7-17
	Controlled Variables	7-18
	The ALLOCATE and FREE Statements for Controlled Variables	7-18
	Stacking Controlled Variables	7-18
	Variable Expressions in Attributes for Controlled Variables	7-19
	The Saving of Variable Extents for Controlled Variables	7-20
	Parameter Variables	7-20
	Variable Expressions in Attributes for Parameter Variables	7-20
	Based Variables	7-20
	The Reference in the BASED Attribute	7-22
	The ALLOCATE and FREE Statements for Based Variables	7-22
	Variable Expressions in Attributes	7-23
	The Saving of Variable Extents	7-23
	The REFER Option	7-24
	Equivalenced Based Variables	7-25
	Simple Based Variables	7-26

		Page
	String Overlay Based Variables.	7-27
	Partial Based Variables	7-28
	Defined Variables	7-29
	Variable Expressions in Attributes for Defined Variables	7-29
	The Saving of Variable Extents for Defined Variables	7-30
	The Uses of Defined Variables.	7-30
	Simple Defining	7-30
	String Overlay Defining	7-31
	Isub Defining	7-31
	Guidelines for the Storage Class.	7-32
	Automatic Variables.	7-32
	Static Variables	7-33
	Controlled Variables	7-34
	Parameter Variables.	7-34
	Based Variables.	7-34
	Defined Variables.	7-35
	The INITIAL Attribute.	7-35
	Initialization Syntax	7-35
	The Use of the INITIAL Attribute.	7-36
	The Capacity of Storage.	7-37
	Conditions for Storage Management.	7-38
	The STORAGE Condition	7-39
	The AREA Condition.	7-39
Section VIII	Expressions	8-1
	General Remarks on Expressions	8-1
	Nested Expressions.	8-2
	Parenthesized Expressions	8-3
	Storage Types of Expressions.	8-3
	Aggregate Expressions	8-5
	Ordering and Optimizing the Evaluation of Expressions	8-5
	Variable References.	8-7
	Preliminary Remarks on Variable References. . .	8-7
	The Major Name in a Variable Reference . . .	8-7
	Simple Variable References.	8-8
	The Form of Simple Variable References . . .	8-8
	The Interpretation of Simple Variable References.	8-9
	Examples of Simple Variable References . . .	8-10
	Subscripted Variable References	8-11
	The Form of Subscripted Variable References.	8-11
	The Interpretation of Subscripted Variable References	8-11
	Examples of Subscripted Variable References.	8-13
	Structure-Qualified Variable References	8-15
	The Form of Structure-Qualified Variable References	8-15
	The Interpretation of Structure-Qualified Variable References	8-16
	Examples of Structure-Qualified Variable References	8-17
	Locator-Qualified Variable References	8-20
	The Form of a Locator-Qualified Variable Reference	8-21
	Associated Storage Types for Locator Values.	8-21
	The Interpretation of Locator-Qualified Variable References	8-22

		Page
	Examples of Locator-Qualified Variable References.	8-23
	Shortened Forms of References	8-25
	Subscript-List Deletion.	8-25
	Examples of Subscript List Deletion	8-26
	Guidelines for Subscript List Deletion.	8-26
	Name Deletion.	8-26
	Examples of Name Deletion	8-27
	Guidelines for Name Deletion.	8-28
	Locator-Qualifier Deletion	8-28
	Guidelines for Locator-Qualified Deletion	8-28
	The Cost of Variable References	8-29
	Constant Literals.	8-29
	Arithmetic Constant Literals.	8-29
	The Form of Arithmetic Constant Literals	8-30
	The Interpretation of Arithmetic Constant Literals	8-31
	Examples of Arithmetic Constant Literals	8-31
	Guidelines for Arithmetic Constant Literals.	8-32
	String Constant Literals.	8-32
	The Form of String Constant Literals	8-32
	The Interpretation of String Constant Literals.	8-33
	Examples of String Constant Literals	8-33
	Attributes for Constant Literals.	8-34
	The Cost of Constant Expressions.	8-34
	Constant References.	8-34
	Statement Constant References	8-35
	The Form of Statement Constant References.	8-35
	The Interpretation of Statement Constant References	8-35
	Examples of Statement Constant References.	8-35
	External Entry Constant References	8-36
	File Constant References.	8-37
	The Form of File Constant References	8-37
	Examples of File Constant References	8-38
	Attributes for Constant Names	8-38
	Programmed Function References	8-39
	The Form of Programmed Function References.	8-39
	The Interpretation of Programmed Function References	8-40
	Examples of Programmed Function References.	8-40
	Built-In Function References	8-42
	The Form of Built-In Function References.	8-43
	The Interpretation of Built-In Function References	8-43
	Examples of Built-In Function References.	8-44
	Differences Between Built-In and Programmed Function References.	8-45
	Operator Expressions	8-46
	The Form of Operator Expressions.	8-47
	The Interpretation of Operator Expressions.	8-47
	The Operator Priority Rules	8-48
	Examples of Operator Expressions.	8-49
Section IX	Operations.	9-1
	General Remarks.	9-2
	Argument Evaluation	9-2
	Nonstandard Operations.	9-3
	Conventions for Examples.	9-3
	The Arithmetic Operations.	9-3

	Page
Conventions for Definitions	9-4
Arguments	9-4
Results	9-5
The Common Data Attributes	9-5
Elementary Operations	9-5
The Prefix Sign Operators	9-6
The Infix Sign Operators	9-6
The Multiplication Operator	9-7
The Division Operator	9-8
The Exponentiation Operator	9-9
The ADD, SUBTRACT, MULTIPLY, and DIVIDE Functions	9-10
Comparison Operations	9-10
The Relational Operators for Arithmetic Values	9-10
The MIN and MAX Functions	9-11
Truncating Functions	9-12
The TRUNC, FLOOR, and CEIL Functions	9-12
The ROUND Function	9-13
The MOD Function	9-14
Sign-Manipulation Functions	9-16
The ABS Function	9-16
The SIGN Function	9-17
Complex Arithmetic Functions	9-17
The REAL and IMAG Functions for Arithmetic	9-18
The COMPLEX Function	9-18
The CONJG Function	9-19
Mathematical Operations	9-19
Conventions for Definitions	9-19
Arguments	9-19
Results	9-20
Evaluation	9-20
Functions Related to Exponentiation	9-21
The EXP Function	9-21
The LOG Function	9-22
The LOG10 and LOG2 Functions	9-22
The SQRT Function	9-23
Trigonometric Functions	9-24
The SIN, COS, and TAN Function	9-24
The SIND, COSD, and TAND Functions	9-25
The Ordinary ATAN Function	9-25
The Ordinary ATAND Function	9-26
The Cartesian ATAN Function	9-26
The Cartesian ATAND Function	9-27
Hyperbolic Functions	9-28
The SINH, COSH, and TANH Functions	9-28
The ATANH Function*	9-29
Functions for Statistical Analysis	9-29
The ERF and ERFC Functions	9-30
String Operations	9-30
Conventions for Definitions	9-30
Arguments	9-31
Concatenate Operations	9-32
The Concatenate Operator	9-32
The COPY Function	9-32
Substring Operations	9-33
The SUBSTR Function	9-33
The INDEX Function	9-34
The BEFORE Function	9-35
The AFTER Function	9-35

	Page
The DECAT Function	9-36
Relational, Length, and Reverse Operations	9-37
The Relational Operators for String Values	9-38
The LENGTH Function	9-40
The REVERSE Function	9-40
Bit-String Operations	9-41
The Logical Operators	9-41
The BOOL Function	9-42
Character-String Operations	9-43
The SEARCH Function	9-43
The VERIFY Function	9-43
The TRANSLATE Function	9-44
Character-Set Operations	9-45
The COLLATE Function	9-45
The LOW Function	9-47
The HIGH Function	9-47
String Functions Defined Elsewhere	9-47
Address and Area Operations	9-48
General Address Functions	9-48
The Relational Operators for Address Values	9-48
The ADDR Function	9-49
The NULL Function	9-50
The NULLO Function*	9-50
An Implementation-Dependent Address Function	9-51
The ADDREL Function*	9-51
An Area Function	9-52
The EMPTY Function	9-52
Array Operations	9-52
Extent Functions	9-52
The LBOUND and HBOUND Functions	9-53
The DIMENSION Function	9-53
Special Array Functions	9-54
The SUM and PROD Function	9-54
The DOT Function	9-55
Conversion Operations	9-55
The Fundamental Conversion Function	9-56
The CONVERT Function*	9-56
Conversion to Arithmetic Data Types	9-58
The REAL Function for Conversion	9-58
The COMPLEX Function for Conversion	9-59
The FIXED Function	9-59
The FLOAT Function	9-60
The BINARY Function	9-61
The DECIMAL Function	9-62
The PRECISION Function	9-63
Conversion to String Data Types	9-63
The CHARACTER Function	9-63
The BIT Function	9-64
Conversion Between Locative Data Types	9-65
The POINTER Function	9-65
The OFFSET Function	9-65
Special Conversion Functions	9-65
The STRING Function	9-66
The UNSPEC Function	9-67
The VALID Function	9-68
Guidelines for Conversion Functions	9-68
System Variable Operations	9-69
System Counter Functions	9-70
The LINENO and PAGENO Functions	9-70
The TIME and DATE Functions	9-70
The Storage Management Functions	9-71

		Page
	The ALLOCATION Function.	9-71
	The SIZE Function*	9-72
	The ON Condition Functions.	9-73
	The ONLOC Function	9-74
	The ONCODE Function.	9-74
	The ONKEY Function	9-75
	The ONFIELD Function	9-75
	The ONCHAR and ONSOURCE Functions.	9-76
	The ONFILE Function.	9-77
Section X	Value Assignment.	10-1
	Preliminary Examples of Assignment Statements. . .	10-1
	Arithmetic Assignment Statements.	10-1
	String Assignment Statements.	10-2
	Address Assignment Statements	10-2
	Area Assignment Statements.	10-3
	Aggregate Assignment Statements	10-3
	The Form of Assignment Statements.	10-4
	Targets	10-4
	The Interpretation of Assignment Statements. . . .	10-5
	Special Restrictions.	10-5
	Overlapping String Targets	10-5
	Area Assignments	10-7
	The Order of Interpretation	10-7
	Pseudo-Variables	10-9
	The REAL and IMAG Pseudo-Variables.	10-9
	The SUBSTR Function as a Pseudo-Variable. . . .	10-10
	The STRING Pseudo-Variable.	10-11
	The UNSPEC Pseudo-Variable.	10-12
	The PAGENO Pseudo-Variable.	10-12
	The ONCHAR and ONSOURCE Pseudo-Variables. . . .	10-13
Section XI	Program Flow.	11-1
	Sequential Execution	11-2
	The IF Statement	11-3
	The Test in an IF Statement	11-4
	The Consequences in an IF Statement	11-4
	The Non-Iterative DO Group as a Consequence	11-6
	The IF Statement as a Consequence.	11-6
	The Dangling ELSE Clause.	11-7
	The DO Group	11-8
	The Iterative DO without Index.	11-9
	The Iterative DO with Index	11-10
	The Single-Value Control	11-11
	The REPEAT Control	11-12
	The FORTRAN Control.	11-13
	The Index of a DO Group.	11-15
	The Non-Iterative DO.	11-16
	The GOTO Statement	11-16
	The GOTO with a Constant Reference.	11-16
	The GOTO with a Non-Constant Reference.	11-17
	The LOCAL Attribute.	11-17
	A Restriction on the Destination	11-18
	Block Execution.	11-19
	Guidelines for Flow of Control	11-19
	Avoidance of Unnecessary GOTO Statements. . . .	11-19
	Layout Conventions.	11-20
Section XII	Procedure Invocation.	12-1
	Arguments and Parameters	12-2
	Argument Classification	12-3

	Page
Examples of Argument Classification.	12-4
Examples of Connected and Unconnected Arguments	12-5
Argument Interpretation	12-6
Examples of Argument Interpretation.	12-6
The Effect of an '*' Extent	12-7
An Example of An '*' Extent.	12-7
Guidelines for Arguments.	12-8
The CALL Statement	12-9
Agreement of CALL with Entry Point.	12-9
The Execution of CALL Statements.	12-9
Interpret Entry Reference.	12-10
Interpret Arguments.	12-10
Activate Procedure	12-10
Execute Procedure.	12-11
Exit From Procedure.	12-11
Examples of CALL Statements	12-12
Function References.	12-14
Agreement of Reference with Entry Point	12-14
The Interpretation of Function References . . .	12-15
Exit From Procedure.	12-15
Fetch Result	12-16
An Example of a Function Reference.	12-16
Procedures	12-17
The PROCEDURE Statement	12-17
The Prefix	12-18
The Parameter List	12-19
The RETURNS Attribute.	12-19
An Example of the RETURNS Attribute . . .	12-20
The RECURSIVE Keyword.	12-20
The OPTIONS Attribute.	12-20
The ENTRY Statement	12-21
An Example of a Multiple-Entry Procedure . .	12-22
The RETURN Statement.	12-22
The END Statement	12-22
Entry References and Declarations.	12-23
Constant Entry References	12-23
Entry Points in the Same External Procedure	12-23
Entry Points in Another External Procedure	12-23
Entry Points in a Non-PL/I Procedure	12-24
Additional OPTIONS Attributes	12-24
Variable Entry References	12-25
An Example of an Entry Variable Reference. .	12-26
Function Entry References	12-27
An Example of an Entry Function Reference. .	12-27
Generic Entry Names	12-28
An Example of a Generic Entry Name	12-28
Recursive Procedure Execution.	12-29
Surface Recursion	12-30
An Example of Recursion Without Arguments. .	12-31
An Example of Recursion With an Argument . .	12-33
An Example of Chained Recursion.	12-34
A Useful Recursive Program	12-35
General Recursion	12-36
Activation Indexes	12-37
Pointers	12-37
An Example of Pointers.	12-38
Parent Designators	12-39
Activation Variable References.	12-40
Statement Address Constant References . .	12-40

		Page
	Setting the Parent Designator.	12-40
	The PROCEDURE Block	12-41
	The BEGIN Block	12-42
	The FORMAT Statement.	12-42
	The ON Unit	12-43
	The GOTO Statement	12-44
Section XIII	Condition Handling.	13-1
	The Principal Features of Condition Handling . . .	13-1
	Conditions	13-2
	Language-Defined Conditions	13-2
	Computational Conditions	13-2
	Storage Conditions	13-3
	Termination Conditions	13-4
	Input/Output Conditions.	13-4
	Programmer-Defined Conditions	13-5
	Condition References	13-5
	Language-Defined Condition References	13-5
	Programmer-Defined Condition References	13-6
	The Declaration of Condition Names	13-6
	Enabling and Disabling Conditions.	13-6
	Condition Prefixes.	13-7
	The Scope of a Condition Prefix	13-7
	Default Enabling and Disabling.	13-8
	An Example of Enabling and Disabling.	13-9
	Establishing and Reverting ON Units.	13-9
	The ON Statement.	13-10
	The REVERT Statement.	13-11
	The Occurrence and Signalling of Conditions. . . .	13-11
	The Occurrence of Conditions.	13-11
	The SIGNAL Statement.	13-12
	Determining the Established ON Unit	13-13
	The ON Unit.	13-13
	Default ON Units.	13-14
	The ON Condition Built-In Functions	13-14
	An Example of Condition Handling	13-16
	Guidelines for Condition Handling.	13-17
	Debugging	13-17
	Enabling Conditions for Debugging.	13-17
	ON Units for Debugging	13-17
	Controlling Exceptional Conditions.	13-17
	Controlling File Communication Conditions. .	13-18
	Controlling Error Conditions	13-18
	Input Data Validation	13-18
	Computational Checks.	13-18
	Resource Management	13-18
	Large Independent Systems	13-18
	General Conditions	13-18
	The ERROR Condition	13-19
	The FINISH Condition.	13-19
Section XIV	Stream Input/Output	14-1
	Stream Data Sets	14-1
	Control Characters in PL/I.	14-2
	Input Streams	14-3
	Output Streams.	14-3
	Pseudo-Streams.	14-4
	Stream Input/Output Files.	14-4
	File-State Blocks	14-5
	File References	14-6
	Stream Input/Output Operations	14-6
	Opening and Closing Files.	14-7

		Page
	The OPEN Statement.	14-8
	The CLOSE Statement	14-9
	Default Files	14-10
	Input/Output Statements.	14-11
	The GET Statement	14-11
	The PUT Statement	14-12
	The FORMAT Statement.	14-13
	Data-Directed Input/Output	14-14
	An Example of Data-Directed Input/Output. . . .	14-14
	A Second Example of Data-Directed Input/Output	14-16
	Principles and Exceptions	14-17
	Guidelines for Data-Directed Input/Output . . .	14-18
	List-Directed Input/Output	14-19
	An Example of List-Directed Input/Output. . . .	14-19
	Compound List Items	14-20
	Pseudo-Variable List Items.	14-22
	Principles and Exceptions	14-23
	Guidelines for List-Directed Input/Output . . .	14-23
	Edit-Directed Input/Output	14-24
	Examples of Edit-Directed Input/Output.	14-24
	Data Format Items	14-26
	String Format Items.	14-27
	String Input.	14-28
	String Output	14-28
	Fixed-Point Format Items	14-28
	Fixed-Point Input	14-29
	Fixed-Point Output.	14-30
	Floating-Point Format Items.	14-30
	Floating-Point Input.	14-31
	Floating-Point Output	14-31
	Complex Format Items	14-32
	Complex Input	14-32
	Complex Output.	14-33
	Picture Format Items	14-33
	Fixed-Point Pictures.	14-34
	Floating-Point Pictures	14-36
	Character Pictures.	14-36
	Control Format Items.	14-37
	The X Format Item.	14-37
	The COLUMN Format Item	14-38
	The SKIP Format Item	14-38
	The LINE Format Item	14-38
	The PAGE Format Item	14-38
	Format Lists.	14-39
	Remote Format Items.	14-39
	Iterated Format Lists.	14-39
	End-Around Repetitions	14-39
	Guidelines for Edit-Directed Input/Output . . .	14-40
	The STRING Option.	14-41
	Conditions for Stream Input/Output	14-42
	The CONVERSION Condition.	14-43
	The ENDFILE Condition	14-43
	The ENDPAGE Condition	14-44
	The NAME Condition.	14-44
	The TRANSMIT Condition.	14-44
	The UNDEFINEDFILE Condition	14-45
Section XV	Record Input/Output	15-1
	Record Data Sets	15-1
	The Organization of Record Data Sets.	15-2
	GCOS Files.	15-2

	Page
Record Files	15-3
File-State Blocks	15-3
File References	15-4
Record Input/Output Operations	15-5
Opening and Closing Files.	15-6
The OPEN Statement.	15-6
File Descriptions.	15-7
The CLOSE Statement	15-8
Keyed Input/Output Operations.	15-8
The Keyed WRITE Statement	15-8
The Keyed READ Statement.	15-9
The Keyed DELETE Statement.	15-9
The Keyed REWRITE Statement	15-10
An Example of Keyed Input/Output.	15-10
Sequential Input/Output Operations	15-11
The Sequential WRITE Statement.	15-11
The Sequential READ Statement	15-11
The Sequential DELETE Statement	15-12
The Sequential REWRITE Statement.	15-12
An Example of Sequential Input/Output	15-13
Based Input/Output Operations.	15-13
Based Input	15-14
An Example of Based Input.	15-14
Another Example of Based Input	15-16
Based Output.	15-17
The Omission of the FROM Option.	15-17
The LOCATE Statement	15-18
Conditions for Record Input/Output	15-18
The ENDFILE Condition	15-19
The KEY Condition	15-20
The RECORD Condition.	15-21
The TRANSMIT Condition.	15-21
The UNDEFINEDFILE Condition	15-21
Appendix A Guide to PL/I Statements.	A-1
Index	i-1

SECTION I

INTRODUCTION

PL/I is a general-purpose, high-level programming language. It is designed for use across the entire spectrum of computer applications, including scientific, business, and system programming; and it is an alternative to FORTRAN, COBOL, or assembly language. The complete PL/I language is a large and complicated language that provides an experienced programmer with unusual power and flexibility. On the other hand, subsets of PL/I can be selected for specific application areas, and these subsets are easy to learn and use.

The language has a wide range of data types and data structures, and these allow program data to be organized in a clear and convenient way. The program syntax and the control statements of PL/I allow programs to be written in a modular, structured, and easy-to-read style.

GCOS PL/I is closely related to a draft standard that is being developed by the American National Standards Institute and the European Computer Manufacturers Association. When this manual mentions Standard PL/I, the reference is to the language described in the draft ECMA/TC10 - ANSI.X3J1 Basis/1, July 1974. This manual does not specify all differences between GCOS PL/I and Standard PL/I.

This manual covers all of GCOS PL/I. Each feature of the language is explained by an example, and the rules of the language are given in definitions that are informal but complete.

Most of this section is devoted to a general description of PL/I. The description begins with a listing of the main features of the language and continues with a consideration of the applications of PL/I. After that, the fundamental notions of program validity and correctness are defined. Finally, several publications that are useful in the study of PL/I are cited.

THE MAIN FEATURES OF PL/I

PL/I brings together in a single language some of the most successful features of earlier programming languages. The design of the language was a major undertaking; it took nearly ten years and involved many separate groups and committees. The final result is a consensus rather than a unified approach to programming. The strength of the language is in the great variety of its features; its weakness is in the relation of these features to one another. A description of the main features of the language follows.

Data Description

In PL/I, a variable is described in terms of the set of values it can accommodate rather than in terms of the hardware storage it occupies. Consider the following declarations:

```
DCL X FIXED DECIMAL(7,2);
DCL Y CHARACTER(8);
DCL Z BIT(36) VARYING;
```

Each of the three variables just declared occupies two words of GCOS memory; therefore, from the point of view of the hardware, the variables are very similar. However, from the point of view of PL/I, each variable is entirely different from the others. The storage designated by X accommodates a fixed-point number with seven decimal digits, two of which occur after the decimal point. The storage designated by Y accommodates a string of eight ASCII characters. The storage designated by Z accommodates a varying length string, with a maximum length of 36 bits.

The handling of data storage in the manner just described is fundamental to the nature of high-level languages. It makes programs less hardware dependent. In addition, it allows the compiler to detect some errors in a program (such as an attempt to take the square root of a program address) and to supply conversions where they are required (as in the assignment of a fixed-point value to a floating-point variable). The more information the data descriptions give, the more efficiently a compiler can manipulate the data.

PL/I is unusual because of the large number of different storage types that are available in the language. A variable can be declared in any of the following ways:

- An arithmetic variable can be declared to accommodate a number with fixed-point or floating-point scale and binary or decimal base. The length of the number can range from one digit to many digits, and the binary point or decimal point can be positioned anywhere in the number. The number can be complex for use in scientific programming. Therefore, virtually all important representations for numbers are available, and a programmer can choose the representation that exactly suits his needs.

- A string variable can be declared to accommodate a sequence of ASCII characters or a sequence of bits. It can be stored as a compact, fixed-length string or a more flexible varying-length string.

- A pictured variable can be used to accommodate a value that can be interpreted either as a number or a character string, as circumstances require. The use of such variables can greatly simplify the formatting of input/output.

- An address variable or an area variable can be used in performing operations that formerly could be performed only in assembly language.

- An array variable can be declared to accommodate a sequence of smaller variables, all of the same type, that are designated by subscripts.

- A structure variable can be declared to accommodate a sequence of smaller variables that are not necessarily of the same type and that are designated by subnames.

Each different kind of data is called a storage type. The large number of storage types in PL/I is the main cause of the size of the language. For each storage type, both the range of values and the representation of values must be defined. For arithmetic and string values, the representation of each value as a character sequence must be defined, so that the value can be read in, printed out, or used as a constant in a program. Rules for conversion must be given for any pair of storage types between which conversion is reasonable. The operators and built-in functions must be defined to operate on many storage types directly, without a preliminary conversion of operands.

A programmer who is learning PL/I needs a casual acquaintance with the full range of storage types. Once the programmer begins to write a specific program, however, he can concentrate on the storage types required by the program. For example, a program can be written that reads in some numbers, performs some computations, and then prints out numbers; such a program requires only arithmetic storage types. Later, a second version might be written that produces output suitable for immediate publication; this version would require character string variables as well as numbers. Still later, the program might be converted to operate on a permanent data base, and address variables and area variables would be useful. An advantage of PL/I is that a program can become more complicated without outgrowing the language.

Program Structure

A PL/I program is a set of one or more external procedures. Each external procedure is compiled separately; nevertheless, any variable can be shared between all the external procedures of a program by declaring it EXTERNAL. This arrangement makes it practical to develop a large program as a collection of separate modules; then, when development is complete, the modules can be used together.

Each external procedure can contain internal procedures. Internal procedures can be nested, so that a given procedure can be programmed in terms of smaller procedures. Each procedure can have its own variables, so that the data as well as the program can be structured by means of nested procedures. This feature of PL/I makes it suitable for top-down program design.

Storage management is closely related to the procedure structure of a program. In most applications, storage management requires little attention from the programmer. When storage management is not specified for a variable, the variable is automatically allocated and freed as control enters and leaves the procedure in which it is declared. Occasionally it is necessary to declare a variable STATIC so that it will remain throughout program execution. Advanced features for storage management are available for use where programmed storage management is necessary.

Computation

The computational power of PL/I is provided by the more than one hundred built-in operations of the language. Each operation is designated either by an operator, such as +, or a built-in function name, such as LOG. The operations are:

- The arithmetic operations, which include the basic arithmetic operators, the relational operators, and built-in functions for comparison, truncation, sign-manipulation, and complex arithmetic
- The mathematical operations, which are built-in functions for exponentiation, logarithms, trigonometry, and statistical analysis
- The string operations, which include operations for putting strings together, taking them apart, comparing them, searching them, and ordering them
- The address and area operations, which are used for advanced methods of storage management and list processing
- The array operations, which are especially for the manipulation of array variables
- The conversion operations, which can be used to perform any reasonable conversion between storage types
- The special operations, which are used for details of input/output, interrupt handling, and the determination of the time and date.

The interpretation of an operation depends on the storage types of its operands; for example, the + operator is interpreted in one way for fixed-point decimal operands, in a very different way for complex floating-point operands, and in yet another way for array operands. It is a fundamental principle of PL/I that if there is a reasonable interpretation for an operator with given operands, then the operator is defined for those operands. Thus in PL/I, as in mathematics, a single operator can have many meanings.

The computation of a value is specified by an expression. Another fundamental principle of PL/I is that an expression can be used wherever a variable could be used to produce a value. Thus an expression can be used to specify the number of elements in an array, the increment of a DO loop, or an output value.

Sequencing

Generally, the statements of a program are executed in the order in which they appear; however, this sequence can be modified by flow-of-control statements, by procedure invocation, or by the occurrence of conditions. Each of these methods for modifying the sequence of execution is considered briefly here.

There are three statements for flow of control. The IF statement causes conditional execution of a statement or a group of statements; the syntax of the statement allows the logic of a program to be laid out in a clear and readable way. The DO statement causes the repeated execution of a group of statements; three different methods are provided for the control of the repetition. The GOTO statement causes unconditional transfer of control.

There are two methods for procedure invocation. A procedure can be invoked by a CALL statement or a function reference. The latter case is of great importance, because it allows a procedure to be called in the midst of the evaluation of an expression and to return a result that is used in the evaluation of the expression.

PL/I has facilities for handling exceptional conditions. Examples of conditions are division by zero, reading of an end-of-file, and use of an array subscript that is out of range. In most cases, condition handling can be left to the built-in mechanisms of PL/I. However, condition handling can be programmed when necessary; for example, a program can be supplied for execution when an end-of-file for a file is read. Furthermore, conditions can be disabled to reduce cost; for example, the subscript range condition can be enabled during debugging of a program but disabled for production.

Input/Output

There are two separate facilities for input/output in PL/I, stream and record input/output. Each is surrounded by its own special purpose features, so the language has many operations in this area.

The facilities for stream input/output are based largely on those of FORTRAN. Stream input/output is intended for dealing with hard copy, such as cards, listings, and print-outs, rather than permanent storage. Statements with the LIST option can be used for input/output with a minimum programming effort. Statements with the EDIT option can be used to lay out and label values in an elaborate way on the pages of a print-out.

The facilities for record input/output are based largely on those of COBOL. Record input/output is used primarily for communication with permanent storage. The statements for record input/output are much simpler than those for stream input/output, and they depend on such related language features as pictured variables.

APPLICATIONS OF PL/I

The use of PL/I in the three major application areas, scientific, business, and system programming is considered here.

Scientific Programming

Many of the features of PL/I are derived from two earlier languages for scientific programming, FORTRAN and ALGOL; in fact, the development of PL/I began with an effort to develop a new version of FORTRAN. Therefore, many of the features of PL/I may be familiar to the programmer with a background in scientific programming.

Most scientific programs can be written using a small subset of PL/I. Such programs are more readable and compact than the corresponding FORTRAN programs would be; and, in GCOS, they are compiled into programs of comparable efficiency. The following guidelines specify a scientific subset of PL/I:

1. Data. Use only the following data types for variables:

   FIXED BINARY(n)
   FLOAT BINARY(n)
   COMPLEX FLOAT BINARY(n)
   CHARACTER(n)
   BIT(1)

   This eliminates a large part of the language: decimal numbers, fractional fixed-point numbers, picture variables, and all of the non-computational variables.

2. Aggregates. Use arrays but not structures. This eliminates the declaration and resolution of structure names.

3. Storage Management. Use only the following storage classes:

   AUTOMATIC
   STATIC
   PARAMETER

   Since the PARAMETER attribute takes care of itself, the choice is really the simple one between the default AUTOMATIC and the occasionally useful STATIC. This eliminates all programmed storage management.

4. Expressions. Use only scalar expressions, and use only simple or subscripted references. This eliminates many unfamiliar features of PL/I.

5. Operations. Use only the following operations:

   arithmetic operations
   mathematical operations
   array operations

   This eliminates more than half of the operations.

6. Condition Handling. Allow conditions to be handled by default except for the ENDFILE condition. This eliminates most uses of the ON statement.

7. Input/Output. Use stream input/output for most input/output. Use the LIST option for easy programming or the EDIT option when the format and layout is elaborate. Use record input/output only for permanent storage of large arrays as GCOS files.

The subset of PL/I just described is a language not much larger than FORTRAN IV. The advantage of PL/I is that a particular application program can grow in complexity without exceeding the limits of the full PL/I language. If features not included in the subset are needed for increasing the efficiency, reliability or capacity of a program, the necessary features are available as part of full PL/I.

Business Programming

PL/I is very different from COBOL, especially in its program structure, computational forms, flow of control, and procedure invocation. Furthermore, certain facilities of COBOL have no counterpart in PL/I; for example, the SORT and MERGE verbs and the report writer. Therefore, PL/I is not easily accepted as a programming language for business applications.

Nevertheless, PL/I was designed to accommodate business programming. It includes generalizations of some of the most successful language features of COBOL, notably picture clauses, structured records, and record input/output. Furthermore, many of its facilities, unfamiliar though they may be, are well-suited to data processing. The success of the method of programming called structured programming has given new impetus to the use of PL/I for business programming because PL/I is well-suited for structured programming and COBOL is not.

Many of the features of PL/I are not required for business programming. The following guidelines specify a business subset of PL/I:

1. Data. Use only the following data types for variables:

   FIXED DECIMAL(m,n)
   CHARACTER(n) VARYING
   CHARACTER(n) NONVARYING
   BIT(1)
   PICTURE"ps"

   This eliminates complex, floating, and binary numbers and all of the non-computational variables.

2. Aggregates. Use both arrays and structures.

3. Storage Management. Use only the following storage classes:

   AUTOMATIC
   STATIC
   PARAMETER

   Since the PARAMETER attribute takes care of itself, the choice is really the simple one between the default AUTOMATIC and the occasionally useful STATIC. This eliminates all programmed storage management.

4. Operations. Use only the following operations:

   arithmetic operations
   concatenation operator and substring functions
   system counter functions

   The arithmetic operations alone are sufficient except where the manipulation of text is necessary.

5. Condition Handling. Allow conditions to be handled by default except for the following:

   ENDFILE
   ENDPAGE
   KEY
   UNDEFINEDFILE

   This eliminates most uses of the ON statement.

6. Input/Output. Use record input/output for all input/output. This eliminates the complicated facilities for stream input/output.

This subset is intended to be a guide for the study of PL/I. It is not intended to restrict the use of PL/I; for example, if a programmer already is familiar with stream input/output he should certainly use it where it is convenient.

System Programming

The most impressive application of PL/I is system programming. PL/I is the only widely available language that permits efficient system programming in a high-level language context. An example of the use of PL/I as a system programming language is close at hand: the GCOS PL/I compiler is written in PL/I and GMAP.

In the following paragraphs, the features of PL/I that are important for system programming are described.

DATA

Any data structure can be described by an appropriate PL/I variable. A table of data can thus be laid out in a natural and convenient way. The packing of data within a table is completely under the control of the programmer; consequently, he is able to define any pattern of bits. He is able to control the size of critical data bases, and can describe such system-dependent data as page tables, interrupt vectors, input/output channel control words, or machine instructions in object programs.

PL/I based variables are valuable in system programming. They provide the programmer with a completely dynamic storage allocation, a powerful form of list processing, and a mechanism for accessing a data base that occupies any given storage location. Through the use of explicitly allocated based structure variables, the PL/I programmer can dynamically create lists, rings, trees, directed graphs, and so on, whose component nodes are based structure variables containing pointers to other nodes.

PROGRAM STRUCTURE

The structure of a PL/I program closely parallels the modular structure of large systems. A PL/I program can consist of several external procedures that call each other, communicating information through argument lists and external variables. An external variable is declared within each procedure that wishes to use it, and all such declarations are equivalent. The variable exists within the address space of the program but is not owned by any procedure of the program.

The system designer can precisely define a module's interface by actually writing PL/I declarations of external variables and procedure entries. By appropriate use of libraries of these declarations and the %INCLUDE macro, the project managers can insure that all the modules use the same declarations of their shared data.

EFFICIENCY

The GCOS PL/I compiler was designed to compile PL/I programs into efficient code. In nearly all cases, storage types can be determined by the compiler. Therefore, most references, conversions, and operations can be compiled as optimized in-line code with very little run-time testing.

The handling of string data is a good example of a language feature that is designed for efficient implementaion. A PL/I character string value is a sequence of characters whose length is unlimited and is determined during program execution. In contrast, a string variable is allocated with a fixed length. The language must reconcile this difference, and the use of string data is somewhat more difficult than it would be if string variables had unlimited length. However, the PL/I treatment of character strings allows the compiler to produce efficient, non-interpretive object code for operations on character strings.

The storage management mechanisms of the language can be implemented efficiently. Static storage can be allocated before execution. Automatic storage requires a stack, which can be implemented by means of a base register at minimal cost. Based storage requires a pair of space management routines that are used when the program calls for allocation or freeing of storage; these routines can avoid garbage collection by using threaded lists to keep track of storage.

The parameter passing mechanism of PL/I was designed to permit compilation of reasonably efficient object code. The calling program always has a declaration of the parameters of the called procedure, even when the called procedure is in a separately compiled part of the program. Therefore arguments can be passed to parameters without any interpretive code.

PROGRAM VALIDITY

The fundamental question of PL/I programming is whether a given character sequence is a valid program. A valid program is one which makes sense according to the definition of PL/I. A program is invalid if the definition of PL/I does not define the result of executing the program. Validity has little to to with whether or not a program does what the programmer wants it to do. A valid program can certainly yield incorrect results; and, sometimes, an invalid program can yield correct results.

Examples of Invalid Programs

The simplest case of an invalid program occurs when the rules of syntax are violated. Consider the following example:

```
P:  PROC;
        DCL SYSPRINT FILE;
        PUT LIST("HELLO")
        END;
```

This example is not a valid program because the statement on the third line does not end with a semicolon.

Most of the syntactic rules of PL/I are easy to learn; in fact, a knowledge of syntax comes automatically from reading examples of valid programs. Furthermore, when a question of syntax does arise, it can be answered in a formal and almost mechanical way by the syntactic formulae that appear in the PL/I Language Manual.

It is possible for a sequence of characters to be a syntactically valid program but not a semantically valid program. Consider, for example, the syntactically valid program:

```
Q:  PROC;
        DCL SYSPRINT FILE;
        DCL X FLOAT;
        PUT LIST(X**3);
        END;
```

In this program, the variable designated by X is not set (assigned a value) before it is used (evaluated). The definition of PL/I says that the value of a variable is undefined until it is set, either by explicit initialization or by assignment. Therefore the value output by the PUT statement is not defined and it follows that the program is not valid.

The rules of PL/I could have been designed so that any syntactically valid program would be completely valid. The undefined cases are not oversights; they were deliberately included to allow more efficient and reasonable interpretation of programs. Consider the example just given, the program Q. If PL/I initialized all variables (say to zero), then X would have a value when it was used in the PUT statement and the program Q would be valid. But in most cases, when a variable is used before it is set, the program is incorrect, regardless of whether the definition makes it valid or not. Thus automatic initialization would be a wasted operation. Certainly the example program seems to be incorrect, because a program that always prints the same value (such as zero) is not useful.

A program can be invalid when it is used with improper input. Consider, for example, the following syntactically valid program:

```
R:  PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL X FIXED DEC(3);
        GET LIST(X);
        PUT LIST(X**2);
       .END;
```

This program is invalid unless it is used with input values whose magnitudes are less than 1000. If a larger value is supplied as input, then the value will not fit in storage and the value of X is not defined.

Observe, once again, that PL/I could have been designed so that the program just given would always be valid. The size of every value assigned to a variable could be checked before assignment, and a specific action could be taken if the value was too large. However, such a check would be costly; and since most assignments are made within a program where values can be controlled, it would be wasteful. Instead of checking every value, PL/I allows the programmer to explicitly call for a check where it is needed. The example just given can be made valid for all input values as follows:

```
R:  PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL X FIXED DEC(3);
(SIZE): GET LIST(X);
        PUT LIST(X**2);
        END;
```

The SIZE condition prefix on the input statement causes the value assigned to X by that statement to be checked. If the value is too large, the system prints an error message and aborts the program, and this is a well-defined action.

The program just given takes a well-defined action for improper input, but the action is drastic. The following program is designed to recover from improper input:

```
R:  PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL X FIXED DEC(3);
        DCL SIZE COND;
        ON SIZE
             BEGIN;
             PUT LIST("TRY AGAIN: ");
             PUT SKIP;
             GOTO L;
             END;
(SIZE):
L:      GET LIST(X);
        PUT LIST(X**2);
        END;
```

This program responds to improper input by performing a programmed action; specifically, it prints "TRY AGAIN: " and calls for a new input value.

Each of the three versions of the program R are useful under appropriate circumstances. If the programmer can be sure that improper input will not be used, the first version is best because it does not entail the extra cost of the size check. If the programmer considers improper input to be a rare and unimportant occurrence, then the second version is best because it is safe but simple. If the programmer considers improper input to be a normal occurrence in the use of the program, the last version is best because it recovers.

The Interpretation of Invalid Programs

The interpretation of an invalid program is not defined for PL/I; however, something happens when an invalid program is compiled and executed. The following cases apply:

- A program that is syntactically invalid is usually detected by the compiler. The compiler prints a diagnostic message and usually declines to produce the object segment.
- Programs that are invalid for non-syntactic reasons are often not detected by the PL/I system, because the cost of checking for such invalid programs is too great. For example, when a variable is used before it is set, some particular value is used; and unless the value happens to make something go wrong elsewhere, program execution proceeds.

The detection of constructs that are invalid for non-syntactic reasons is a major part of the debugging of a program.

SUGGESTIONS FOR THE STUDY OF PL/I

The following paragraphs describe various publications that are designed for the study of PL/I. The description begins with a list of three introductory texts, continues with remarks on this manual, and concludes with notes on related Honeywell publications.

Introductory Texts

An introductory text on PL/I provides examples and a framework for further study. Three useful texts are:

1. PL/I Programming Primer, by Gerald M. Weinberg, McGraw Hill, 278 pages. This short book provides a smooth and efficient introduction to PL/I. It keeps strictly to the subject of elementary PL/I, and can be read in a short time. The use of a single programming problem throughout most of the book provides continuity; and the example program can later be run as a GCOS PL/I program.

2. PL/I for Scientific Programmers, by C. T. Fike, Prentice-Hall, 241 pages. This book is recommended for the experienced FORTRAN programmer. Most chapters end with a section called "PL/I and FORTRAN" and thus emphasize the relationship between the two languages. The book is organized according to the features of the language rather than having a narrative form; therefore, it cannot be read as quickly as Weinberg's book.

3. An Introduction to Programming, by Richard Conway and David Gries, Winthrop Publishers, Cambridge, Massachusetts, 460 pages. This book is a complete course in high-level programming that is based on PL/I. It is too long for a quick reading, but it provides good background reading because its examples are all in PL/I. Included are discussions of top-down program development, confirmation of program correctness, recursive programming, scientific programming, and file processing.

There are many other introductory texts for PL/I, but most of them have the disadvantage that they go into details of specific computer hardware and specific operating systems, topics that do not contribute to the study of GCOS PL/I.

The PL/I Reference Manual

The study of PL/I should continue with an examination of this manual. The object of this examination should be to become familiar with the general features of GCOS PL/I and the way these features are described in this manual.

A variety of techniques are used in this manual to assist the reader in understanding PL/I and writing efficient programs. Examples are given in abundance and are often designed to cover all possible cases of a feature. Principles of design of PL/I are given to provide a framework for the details of the language. Guidelines for efficient and clear programming are given when, as is often the case, the language allows more than one way of programming a particular operation. Finally, repetition is freely used to avoid the use of cross references and to emphasize the features of the language that are most important.

THE ROLE OF EXAMPLES IN THIS MANUAL

Many of the examples in this manual are complete programs. The use of complete programs has several advantages. First, such examples provide a guide for the details of programming style by showing how programs should be laid out for readability, how abbreviations should be used, and, generally, how statements should be put together to form a program. Further, an example that is a complete program does not require a discussion of the context of the example; it is, by definition, complete. Finally, the complete example programs can be used by the reader to experiment with the execution of PL/I programs in the GCOS environment.

However, most of the examples are not realistic applications of PL/I. A realistic PL/I program, even a small one, requires several pages of background and explanation, and such a discussion is beyond the scope of this manual. The examples in this manual are designed to show how certain statements are executed, not how they should be used to solve problems. Sometimes the absence of realism is obvious, as with an example program that does nothing but read two numbers and print out their sum. In other cases, the examples are complicated; nevertheless, these examples are usually contrived in order to illustrate an important feature of PL/I rather than to solve a real problem.

THE ROLE OF GUIDELINES IN THIS MANUAL

PL/I provides many ways to program the solution of any given problem. Usually it is not sufficient to write a program that produces the correct results; in addition, the program must be reasonably efficient and fairly reasonable. The guidelines given in this manual are designed to assist programmers in choosing among the many options offered in PL/I. When a reader has guidelines that are better for his own purposes than those given in this manual, he should ignore those given here.

In some places in this manual, guidelines are given under a special heading, such as "Guidelines for Arithmetic Data Types"; in other places, guidelines are mentioned as part of the definition of a feature. Usually, a guideline is distinguished by the use of the word "should" instead of "must"; for example, "when a storage unit is used for an exact integer, it should be FIXED BINARY."

THE ORGANIZATION OF THIS MANUAL

The manual can serve both as an advanced tutorial and an informal reference. The sections are organized to reflect the main features of the language. After the introductory section, three aspects of data values are considered: values as abstractions, values in storage, and the conversion of values from one storage type to another. The sections that cover this material are:

II. Values
III. Value Storage
IV. Value Conversion

Next, the overall syntax of a program is considered, ranging from small constructs (such as the identifier) to intermediate constructs (the statements) to large constructs (the blocks). Against this background, the declaration of identifiers and the management of storage is described. The sections are:

V. Program Syntax
VI. Declarations
VII. Storage Management

Next, the features that are used to compute and store values are considered. The sections are:

VIII. Expressions
IX. Operations
X. Value Assignment

Next, the features that are used to determine the sequence in which program statements are executed are described. The sections are:

XI. Program Flow
XII. Procedure Invocation
XIII. Condition Handling

Next, the statements for input/output are described. The sections are:

XIV. Stream Input/Output
XV. Record Input/Output

The manual has an appendix that gives the syntax of all of the statements of PL/I, except the DEFAULT statement.

Related Manuals

Additional information on the writing and execution of GCOS PL/I programs is available in the following Honeywell publications:

1. The PL/I Language Manual (Order Number AG94) is a semi-formal definition of Multics PL/I. Since Multics PL/I and GCOS PL/I are essentially the same, this manual can be used to obtain answers to specific questions about the syntax of the language.

2. The GCOS PL/I User's Guide (Order Number DE04) describes running PL/I programs in the GCOS environment. It is intended to provide sufficient information about control cards, file attachment, and system interfaces for the compilation and execution of PL/I programs.

The PL/I Language Manual is the ultimate authority on Honeywell PL/I. For the programmer who is beginning a study of PL/I, the manual may be difficult to use and understand because it is designed to be a definition, not an explanation, of PL/I. To the programmer who has learned Honeywell PL/I, however, the Language Manual is the source of all necessary information about Honeywell PL/I.

The PL/I Language Manual describes the meaning of any given program, but it does not explain how to write a program. Therefore, it is essential to approach the manual with a well-formulated question, a question that is usually in the form "what does the following PL/I construct mean: ...?" If it is impossible to come up with such a question, then the appropriate section of this manual should be used to get the rules, examples, or guidelines necessary to formulate a question.

SECTION II

VALUES

The most important feature of PL/I is the great variety of the values that are provided. When a PL/I program performs a calculation, whether business or scientific, it does so in terms of arithmetic values. When a program accepts input from the user or prints out a listing of results, it transmits string values. When a program refers to the storage for its own data or to the code for its own statements, it uses address values. When a program needs to control storage management, it uses an area value. And when a program needs to treat several values as a single entity, it groups the values into an aggregate value.

In this section, each kind of value is described and the operations that can be applied to the value are summarized. The values are described in an abstract way, without consideration of the way in which they are stored in memory. The purpose of this section is to indicate the computational power of PL/I.

ARITHMETIC VALUES

Standard PL/I does not specify the range of the arithmetic values; instead, it leaves this choice to each implementation of PL/I. Naturally, a useful implementation must provide for a reasonably large set of values. The range provided by GCOS PL/I is very large indeed, and is as follows:

- The greatest magnitude of a GCOS PL/I value is (10**186 - 10**127). This value is expressed as a decimal number by writing 59 9 digits followed by 127 0 digits. The number would occupy about three lines of uninterrupted decimal digits; and its value far exceeds any measurement encountered in business or scientific applications.

- The smallest magnitude of a value is (10**-128). This value is expressed as a decimal number by writing a decimal point followed by 127 0 digits followed by a 1. Again, this number would occupy several lines and its magnitude is much smaller than any measurement encountered in practice.

- The precision of arithmetic values is such that two values can differ by as little as one part in 10**59. This is the same as saying that a value can be expressed as an exact 59-digit decimal number followed by a suitable scale factor. Thus PL/I can supply a very good approximation to any given number.

The arithmetic values just described are called real values, and real values are the only ones used in business applications. PL/I also supplies complex values. Each complex value is composed of a pair of real values, and thus the range of complex values is determined by the range of the real values. Complex values are used only in scientific applications.

A number can be expressed exactly as a PL/I arithmetic value only if it can be expressed exactly in decimal positional notation. For example, 73/2 can be expressed exactly as 36.5 and 13/320 can be expressed exactly as 0.040621875; so these are PL/I arithmetic values. On the other hand 1/3 and the square root of two cannot be expressed exactly this way, and so must be approximated. If a good approximation is acceptable, then GCOS PL/I can provide that approximation; otherwise, special techniques must be applied.

PL/I has many operations for the manipulation of arithmetic values. The familiar arithmetic operators are present; in addition, several dozen built-in functions for integer arithmetic, trigonometry, logarithms, and statistics are provided.

STRING VALUES

A string value is a sequence of characters. Standard PL/I does not specify the set of characters that can appear in a string; instead, the choice of characters is left to each implementation of PL/I. The character set used in GCOS PL/I is the ASCII character set, given under "The COLLATE Function" in the section on "Operations". It is composed of 128 characters, as follows:

- 52 characters that represent the letters of the alphabet in both lower and upper case
- 10 characters that represent the decimal digits
- 18 characters that represent most of the special symbols on a typewriter keyboard
- 14 characters that represent special symbols not found on an ordinary typewriter keyboard, many of which are useful in writing mathematical expressions or non-English text
- 1 character for the blank
- 33 characters that are nonprinting control characters including, for example, horizontal tab, new line, new page, and carriage return

Because this character set includes characters that control the layout of the pages, a printed document of many pages can be composed, stored, and edited as a single PL/I character string value.

The number of characters in a character string value is the length of the value. The maximum length allowed in GCOS is 256K.

Several operators and built-in functions are provided for the manipulation of string values. String values can be concatenated and a substring of a given string value can be extracted. The equality operators ('=' and '^=') can be applied to strings. A collating sequence orders the character set, so it is possible to apply inequality operators (such as '<') to strings. Other functions are available for use in such advanced applications as command interpretation and compiler construction.

A special kind of string value is recognized; namely, a string value composed only of the bits 0 and 1. Such a string value is a bit-string value. When a bit-string value contains a single bit, it is a Boolean value; and it represents "true" or "false" depending on whether the single bit is 1 or 0. By extension, a bit-string value of length greater than one can be treated as a sequence of true/false indicators. The familiar Boolean operators are provided for use with the bit-string values.

Any arithmetic value can be converted to a string value; the result is a conventional representation of the arithmetic value. Conversely, a string can be converted to an arithmetic value, but only if the string value is a valid representation of an arithmetic value. These conversions are essential because a value must be expressed as an arithmetic value if an arithmetic operation is applied to it, but must be expressed as a string value before it can appear in the output stream. In addition to providing functions that perform these conversions in a straightforward way, PL/I has pictured character-string variables, which allow numeric values to be maintained as character-string values in an automatic and convenient way.

ADDRESS VALUES

Every PL/I statement in a program has a unique address, and every data storage unit also has a unique address. It is common in computing to think of an address as an integer and thus endow it with arithmetic properties; but as a value in PL/I, an address has no other property than its association with a particular program statement or storage unit. For example, the "next" address is not defined for any address, so addresses are not even ordered.

The manipulation of address values in PL/I is deliberately limited. An address value can be assigned to a variable and can be produced by a reference to a constant, a variable or a function. The operators '=' and ' =' can be used to determine whether or not two address values are identical. The only conversion that can be applied to address values is that between a pointer value and an offset value.

There are three kinds of address values: statement, locator, and file values; and descriptions of these follow.

Statement Values

A statement value designates a statement in a program. The value is classified as a label, entry, or format value according to the following rules:

- A label value designates a statement to which control can be transferred by a GOTO statement. Any statement except a PROCEDURE, ENTRY, or FORMAT statement can be designated by a label value.
- An entry value designates a statement to which control can be transferred by a CALL statement or a function reference. There are two such statements, the PROCEDURE statement and the ENTRY statement.
- A format value designates a statement that can supply a format list to a stream input/output statement. There is one such statement, the FORMAT statement.

Locator Values

A locator value designates a storage unit for program data. There are two types of locator values, as follows:

- A pointer value is used by itself to access a storage unit.
- An offset value is used in conjunction with an area name to access a storage unit.

The pointer value is similar to the absolute machine address used in assembly language programming, and the offset value is similar to a relative or based machine address. Conversion between a pointer value and an offset value can be performed relative to a given area value.

File Values

In GCOS PL/I, a file value designates a collection of stored values called a file-state block. The values in the file-state block record the status of a data set that is currently being used for input/output. The file value is a more specialized kind of address value than either the program address value or the locator value.

AREA VALUES

An area value is an ordered set of PL/I values. The entire set of values can be assigned to an area variable, passed as a procedure argument, returned as a procedure result, or transmitted as input or output. However, the principal use of an area value is as part of a specialized mechanism for the efficient management of storage. The most important operations on an area are the allocation and freeing of storage for new values within the area. However, the details of these operations are not relevant to this description of values. The area value plays a role in the definition of the OFFSET value, already mentioned under "Locator Values".

AGGREGATE VALUES

An aggregate value is an ordered set of PL/I values. Because it is a set of values, it provides a way of handling a collection of values as a single computational entity. Because it is ordered, it has a first component, a second component, and so on. And because it can contain any PL/I values, it can have another aggregate as a component. Thus an aggregate value can be used to collect values together and arrange them in a hierarchical manner, with aggregates within aggregates.

There are two kinds of aggregate, the array and the structure. An array must contain values that are all of the same kind, whereas a structure is not restricted in this way. Aside from this, the two kinds of aggregate differ in the way they are stored and referenced.

Most of the operations of PL/I can be applied to aggregate values. Each operation is not individually redefined for this purpose; instead, a general rule is applied: the ith component of the result of the operation is produced by applying the operation to the ith component of each of the given operands. Thus the aggregate operation is defined in terms of the conventional, non-aggregate, operation.

THE CLASSIFICATION OF VALUES

The previous paragraphs have traced a progression from the mathematical values of computation to the specialized values required for efficient programming. The arithmetic values are those universally accepted for use in calculation. The string values are used for objects of universal importance (printed pages); but their precise and restricted definition is characteristic of computer programming rather than conventional usage. The address values are meaningful only in relation to the program statements or stored data, and so are creatures of computer programming. The area values represent a further descent into the special techniques of efficient programming. The aggregate values appear to swing back toward the world of mathematics; but their usage is oriented toward programming rather than mathematics.

The following list is a hierarchy for the PL/I values described in this section. Some useful supplementary terminology has been introduced.

scalar: any PL/I value that is not an aggregate
  computational: a value of interest beyond programming
    arithmetic: a number
    string: a sequence of ASCII characters
      character string: an unrestricted string
      bit string: a string of zero's and one's
      pictured character-string: a specially restricted string
  non-computational: a value of interest only for programming
    address: a value that designates a statement or a storage unit
      statement: the address of a statement
        label: the destination of a transfer
        entry: a PROCEDURE or ENTRY statement
        format: a FORMAT statement
      data: the address of data
        locator: the address of storage for a value
          pointer: an "absolute" address
          offset: a "relative" address based in an area
        file: the address of a file-state block
    area: values gathered together by storage management
aggregate: an ordered set of values
    array: components must have the same data type
    structure: components may differ in data type

SECTION III

VALUE STORAGE

The notion of values in the abstract, as described in the preceding section, is the link between the computing requirements of the outside world and the specialized technology of computers. In contrast, the notion of values in storage, as described in this section, is dependent on features of computers that have no relevance to the outside world except for their contribution to efficient, low-cost computing. Binary numbers, normalized floating-point values, limited character sets, and rigid limitations on precision are features of computer hardware that do not have counterparts in the conventional methods of pencil-and-paper computing.

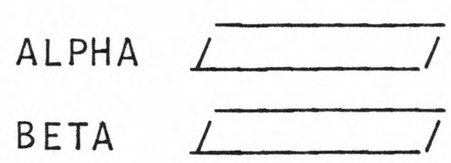
A principal objective of the designers of PL/I was to provide a language in which efficient use of the computer hardware was possible. Therefore, PL/I allows a programmer to give a detailed description of the storage used for each value that is generated during program execution. Since there are many ways to store a value in computer memory, there are many types of value descriptions in PL/I; in fact, the size and complexity of PL/I is largely a consequence of the many ways in which value storage can be described.

As this section continues, it gives a brief description of storage units, which are used to hold PL/I values during program execution. After that, the section gives a long description of the storage types, which are used to describe the kinds of values accommodated by a given storage unit. Most of the description of the storage types follows the same outline as the previous section on "Values", dealing with arithmetic, string, address, area, and aggregate values; however, a new concept, alignment, is described at the end of the section.

STORAGE UNITS

Whenever a program refers to a value in any way, that value resides in a storage unit. When a constant appears in a program, it refers to a storage unit that contains an unchanging value. When a variable name appears, it refers to a storage unit that is used to store a computed result for later use. Even a function reference or an operator expression designates a storage unit in which its result is stored, briefly, until that value is used elsewhere.

Suppose that the variable names ALPHA and BETA are used in a PL/I program; then a portion of data storage can be diagrammed as follows:

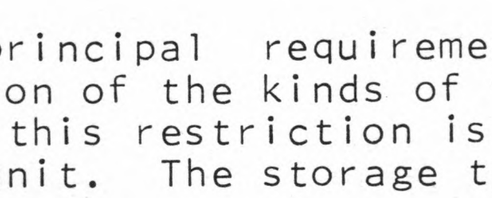


This diagram contains two storage units. Each storage unit consists of a designator, which is the name of a variable, and a box, which can hold a value. Suppose, next, that the following assignment statements are executed:

```
ALPHA = 3.8;

BETA = -2*ALPHA;
```

Immediately after these assignments, the storage units become:



Each storage unit now has a contents as well as a designator and a box.

STORAGE TYPES

It is possible to design a programming language in which each storage unit can hold any type of value. Some of the interactive languages for the solution of simple problems by non-programmers are designed in this way. In such a language, the same variable can accommodate any kind of number or, for that matter, an array of 50 numbers; and if the language has string values or address values, then the variable can also accommodate them. Such a language is easy to learn, but its programs are executed much less efficiently than they could be.

A principal requirement for the efficient execution of a program is the restriction of the kinds of values that can be assigned to a given storage unit. In PL/I, this restriction is applied by associating a storage type with each storage unit. The storage type gives three kinds of information, as follows:

- The data type describes the range and representation of storage for a single datum, such as a number or a character-string or a program address.
- The aggregate type describes the way storage for a collection of values is arranged in an array or a structure.
- The alignment type describes the way the storage is laid out in hardware memory and thus determines the memory required and the ease of access.

PL/I provides for efficiency in two ways. First, it requires a storage type so that the range of the storage unit is known when the program is compiled. Second, it provides many different storage types so the programmer can choose the representation best suited to the problem.

Two introductory examples of the interpretation of storage types follow. The examples depend on rules that are given later in this section, when the various data types, aggregate types, and alignment types are described.

As the basis for the first example, consider the following DECLARE statement:

```
DCL GAMMA FIXED(8);
```

This statement gives the storage type for the variable named GAMMA; it does so by means of the scale attribute FIXED and the precision attribute (8). The statement uses various abbreviations and defaults for the storage type, and without them the statement would be:

```
DCL GAMMA REAL FIXED BINARY PRECISION(8,0) ALIGNED;
```

According to this declaration, the storage type for GAMMA is as follows:

- The data type is REAL FIXED BINARY PRECISION(8,0). This means that the contents of the storage unit named GAMMA must be a number that can be expressed as a signed eight-digit binary integer.
- The aggregate type is scalar since no aggregate type is explicitly given in the DECLARE statement. This means that the storage unit accommodates a single number, not an array or a structure of numbers.
- The alignment type is ALIGNED. In this case, a full word must be allocated for the variable, so that access to the variable is efficient.

The details of the data type for this example are discussed later in this section under "Arithmetic Storage".

The following is a diagram of the storage unit for GAMMA as just declared:

```
          FIXED(8)
GAMMA   /   -15    /
```

The storage unit now has the storage type written above it as well as a designator, a box, and a contents. Observe that the contents is a value that satisfies the restriction imposed by the storage type.

As a second example of the declaration of the storage type, consider the following:

```
DCL 01 CUSTOMER ALIGNED,
       02 NAME CHAR(18),
       02 CODE(2) DEC(4);
```

This statement gives the storage type for the variable named CUSTOMER; it does so by means of attributes and level numbers. The storage type, by itself, is:

```
01 ALIGNED, 02 CHAR(18), 02 DIM(2) DEC(4)
```

Once again, the statement uses various abbreviations and defaults for the storage type, and without them the statement would be:

```
DCL 01 CUSTOMER ALIGNED,
       02 NAME CHARACTER(18) NONVARYING ALIGNED,
       02 CODE DIMENSION(2) REAL FIXED DECIMAL PRECISION(4,0) ALIGNED;
```

According to this declaration, the variable CUSTOMER is a sequence of three components, and its storage type is as follows:

- The data type is CHARACTER(18) NONVARYING for the first component and REAL FIXED DECIMAL PRECISION(4,0) for the last two components. This means that the first component accommodates a string of 18 characters and the second and third components each accommodates a signed four-digit decimal integer.

- The aggregate type is 01, 02, 02 DIMENSION(2). This means that the variable is a structure with two members, and the first member is a scalar while the second is a one-dimensional array with two elements.

- The alignment type is ALIGNED for all three components. This means that the variable should be laid out in memory to permit efficient access rather than to save space. In GCOS, the variable takes nine words; if it had been UNALIGNED, it could have been packed into seven words.

The details for this storage type are given later in this section; the purpose of the example is to provide an introduction, not a complete explanation.

The following is a diagram of the storage units for CUSTOMER as just declared:

```
                                 CHAR(18) ALIGNED
CUSTOMER   .NAME          /______________________/

                                 DEC(4) ALIGNED
--------   .CODE(1)       /______________________/

                                 DEC(4) ALIGNED
----------- (2)           /______________________/
```

In this diagram, the aggregate type is shown by the way the three designators are arranged. The hyphens are used like ditto marks, so that the designators for the storage units are:

```
CUSTOMER.NAME
CUSTOMER.CODE(1)
CUSTOMER.CODE(2)
```

For convenience of discussion, the storage for the entire variable is called a storage unit. Thus one speaks of a structure storage unit that is made up of three component storage units just as one speaks of a structure value made up of three component values.

ARITHMETIC STORAGE

PL/I is designed primarily for operations on arithmetic values. If differences of scaling and precision are ignored, there are eight different ways to represent an arithmetic value in storage. The choice of one of these kinds of storage and the choice of an appropriate precision is a choice between convenience and efficiency. For example, it is more convenient to use decimal numbers throughout, but scientific computations can be performed much more efficiently in binary. For another example, it is more convenient to use a large number of digits for a variable, but it is more efficient to determine exactly how many digits are required and use no more. The selection of the type of storage for an arithmetic variable is an important part of the engineering of a PL/I program.

This discussion begins by defining the arithmetic data types and their corresponding storage units, continues with abbreviations and defaults that allow data types to be written more concisely, gives examples of various arithmetic storage units, and concludes with guidelines for the selection of a data type for a given purpose.

Arithmetic Data Types

The complete data type for an arithmetic storage unit is given as a sequence of four arithmetic attributes. These attributes are the mode, scale, base, and precision, as described in the following paragraphs.

THE MODE ATTRIBUTE

The mode attribute is one of the following keywords:

```
REAL
COMPLEX
```

These attribute keywords are taken from the language of mathematics where they have a technical meaning that is different from their everyday meaning. The choice of mode is important only in scientific applications that make use of the theory of complex numbers. In all other applications of PL/I, the mode attribute must always be REAL.

A storage unit with the REAL attribute accommodates a number that can be represented as a signed sequence of digits in which a decimal or binary point appears. This includes all numbers that are used in business applications, system programming, and everyday calculation. It also includes most numbers used in scientific applications.

A storage unit with the COMPLEX attribute accommodates a number that can be represented as a pair of real numbers, called the real part and the imaginary part, respectively. Both members of the pair have the same scale, base, and precision, as specified by the other attributes of the data type.

THE SCALE ATTRIBUTE

The scale attribute is one of the following keywords:

```
FIXED
FLOAT
```

These attribute keywords refer to the decimal or binary point of the number. In a FIXED storage unit, the point cannot move, whereas in a FLOAT storage unit, the point can be thought of as moving to accommodate a wider range of values in a given number of digits.

A storage unit with the FIXED attribute accommodates a value that can be represented as a signed sequence of digits in which a decimal or binary point appears. The point can appear anywhere, but its position is determined when the storage unit is created and remains fixed throughout the existence of the storage unit. For a given FIXED storage unit, the number of digits and the position of the point are both specified by the precision attribute, which is described a little later.

A storage unit with the FLOAT attribute accommodates a value that can be represented in one of the following forms:

m * (2**e) (if the scale is BINARY)

m * (10**e) (if the scale is DECIMAL)

where m is the mantissa and e is the exponent. The mantissa is a signed sequence of digits in which a point appears. For BINARY scale, the point appears to the left of the first digit, so that the mantissa is a fraction. For DECIMAL scale, the decimal point appears to the right of the last digit, so that the mantissa is an integer.

The number of digits in the mantissa is determined by the precision attribute, which is described later. The exponent must lie in the range:

$-128 \leq e \leq +127$

When a value is assigned to the storage unit, there is some freedom in the choice of the mantissa and the exponent. For example, if the mantissa has four decimal digits, then the value 1.4 can be represented in the following ways:

```
+0014.*(10**-1)
+0140.*(10**-2)
+1400.*(10**-3)
```

However, a representation is never chosen that discards more low-order digits than necessary, and 1.4 would not be represented as

```
+0001.*(10**0)
```

Thus the variable exponent not only allows a wide range of values but also allows the full use of each digit of the mantissa whenever necessary.

The exponent can be thought of as expressing the number of places the mantissa point should be moved to give the true value of the storage unit. That point of view is the source of the term "floating point" and the keyword FLOAT.

THE BASE ATTRIBUTE

The base attribute is one of the following keywords:

```
BINARY
DECIMAL
```

These attribute keywords refer to the number system that is used in representing the value.

A storage unit with the BINARY attribute uses binary digits in representing its value. Thus a FIXED BINARY storage unit that has three significant digits followed by a binary point can accommodate the integers from -7 to +7.

A storage unit with the DECIMAL attribute uses decimal digits in representing its value. Thus a FIXED DECIMAL storage unit that has three significant digits followed by a decimal point can accommodate the integers from -999 to +999.

THE PRECISION ATTRIBUTE

The complete precision attribute has one of the following forms:

PRECISION(p,q)

PRECISION(p)

where p and q are the number-of-digits and the scale-factor, respectively. The first form is used when the scale is FIXED and the second form is used when the scale is FLOAT. The number-of-digits and the scale-factor must each be given as an optionally-signed integer constant. They cannot be given as general expressions because their values must be known to the compiler.

The number-of-digits determines how many digits appear in the storage unit. The number-of-digits must lie within a certain range, and that range depends on the scale and base attributes that appear with the precision attribute. The ranges are:

Scale and Base	Minimum	Maximum
FIXED BINARY	1	71
FLOAT BINARY	1	63
FIXED DECIMAL	1	59
FLOAT DECIMAL	1	59

This table shows, for example, that the storage type REAL FIXED BINARY PRECISION(70,0) is valid but REAL FIXED DECIMAL PRECISION(70,0) is not. The ranges are not part of Standard PL/I; they are chosen for each implementation, and the values shown here are those for GCOS.

In a FLOAT storage unit, the number-of-digits refers to the digits in the mantissa and does not include digits used in the exponent. In the case of a FLOAT BINARY storage unit, the number-of-digits establishes a minimum for the number of mantissa digits. The purpose of this latitude is to permit efficient use of floating-point binary hardware; in GCOS, the provision is exploited as follows: if the number of digits is 27 or less, then 27 digits are used for the mantissa; otherwise, 63 digits are used. This corresponds to the choice between single and double precision floating-point binary in the GCOS hardware.

The scale-factor determines the number of significant digits to the right of the point in a FIXED storage unit. The scale-factor is restricted to the following range:

Scale	Minimum	Maximum
FIXED	-128	+127
FLOAT	(not applicable)	

A fixed-point storage unit can have filler zeros. These zeros are used to position the point and are not counted in the determination of the number-of-digits. Suppose the precision attribute is PRECISION(p,q); then three cases must be considered:

- If $p \geq q \geq 0$, then no filler zeros are required because the point is adjacent to one of the significant digits. For example, the storage type REAL FIXED DECIMAL PRECISION(5,2) accommodates any value that can be represented by a sign, three significant digits, a decimal point, and two more significant digits. Thus the value -203.49 or the value 1.2 can be accommodated.

- If $p < q$, then q-p filler zeros are assumed between the point and the first significant digit. For example, the storage type REAL FIXED DECIMAL PRECISION(5,7) accommodates any value that can be represented by a sign, a decimal point, two filler zeros, and five significant digits. Thus the fraction .0044244 is accommodated by this storage type but .0144244 is not.

- If $q < 0$, then -q filler zeros are assumed between the last significant digit and the point. For example, the storage type REAL FIXED DECIMAL PRECISION(5,-3) accommodates any value that can be represented by a sign, five significant digits, three filler zeros, and a decimal point. Thus the integer 55955000 is accommodated by this storage type, but the integer 55955300 is not (and must be approximated).

In practice, most programs use only fixed variables whose precision attribute satisfies $p \geq q \geq 0$; therefore filler zeros are not often used.

ABBREVIATIONS AND DEFAULTS

A typical program uses many variables, and a data type must be given for each variable; therefore, PL/I permits the use of many abbreviations and defaults in the specification of data type attributes.

From the point of view of the PL/I compiler, any selection of abbreviations and defaults can be used, and the selection can differ from one place in the program to another. However, the inconsistent use of abbreviations and defaults makes a program confusing, and a consistent policy should be adopted. This Reference Manual provides one such policy: every abbreviation or default of PL/I should be used except for those whose avoidance is explicitly recommended in this Reference Manual.

The following abbreviations are defined for the keywords used in arithmetic attributes:

Keyword	Abbreviation
BINARY	BIN
DECIMAL	DEC
PRECISION	PREC (rarely used)
PRECISION	(omit the entire keyword PRECISION if it immediately follows a mode, scale, or base attribute)
COMPLEX	CPLX (not recommended)

The abbreviation PREC is rarely used because it is customary to write the precision attribute immediately after some other arithmetic attribute and omit the entire PRECISION keyword. For example, REAL FIXED DECIMAL PRECISION(5,2) is abbreviated as REAL FIXED DEC(5,2) so that all that is left of the precision attribute is (5,2). The abbreviation CPLX is not recommended because it is difficult to remember and impossible to pronounce.

A default attribute is the attribute that is assumed by the PL/I compiler when a required attribute is not given in the program. For example, if a variable is declared FIXED DEC(5,2), then the compiler treats the variable as if it had been declared REAL FIXED DEC(5,2); it does so because the default attribute for the mode is REAL. The defaults for the arithmetic data type are:

Omitted Item	Default
mode attribute	REAL
scale attribute	FIXED
base attribute	BINARY
number-of-digits	17 (for FIXED BINARY) 7 (for FIXED DECIMAL) 27 (for FLOAT BINARY) 10 (for FLOAT DECIMAL)
scale-factor	0 (for FIXED scale only)

The table just given shows that the default value for the number-of-digits depends on the scale and base of the data type; thus there are four possible default values. The default values for the number-of-digits are not part of Standard PL/I, and the values given are those for GCOS. However, it is understood that any implementation of PL/I should choose a default for FIXED values that is appropriate for an index or subscript and a default for FLOAT values that is appropriate for most scientific calculations.

The abbreviations and defaults are designed to favor the most commonly used storage types, as shown in the following examples:

Complete Data Type	Recommended Form
REAL FIXED BINARY PRECISION(17,0)	FIXED
REAL FIXED BINARY PRECISION(30,0)	FIXED(30)
REAL FIXED DECIMAL PRECISION(8,2)	DEC(8,2)
REAL FLOAT BINARY PRECISION(27)	FLOAT
REAL FLOAT BINARY PRECISION(60)	FLOAT(60)
COMPLEX FLOAT BINARY PRECISION(27)	COMPLEX FLOAT

The first two examples require some discussion. Since FIXED is the default scale, it could have been omitted from the first example, leaving no attributes at all. That is valid, but a declaration with no vestige of a data type is confusing. The second example.could have been written in two other ways:

```
BIN(30)
PREC(30)
```

However, the use of FIXED contrasts nicely with the use of FLOAT in the fourth and fifth examples, and is most consistent with familiar usage.

Examples of Arithmetic Storage Units

As an example of the use of an arithmetic storage unit, consider the following program:

```
P:  PROC;
        DCL ALPHA DEC(6,2);
        ...
        ALPHA = -31.253;
        ...
        END;
```

The storage unit for ALPHA can be diagrammed as:

```
          DEC(6,2)
ALPHA   / -31.25  /
```

The data type is written above the box to indicate the restriction on the value accommodated by the box. Since the scale-factor is two, the storage unit can accommodate only two fractional digits; therefore, the value -31.253 is approximated when it is assigned to the storage unit.

A diagram called a data frame is useful in describing the capacity of a storage unit. A data frame is produced by combining the data type with the box; the result is a diagram that suggests the structure of storage. When a data frame is used for the variable ALPHA just discussed, then the result is:

```
          S 9 9 9 9   9 9
ALPHA   /-/0/0/3/1/./2/5/
```

In this diagram, each of the seven boxes holds a single character. The symbol above each box determines the kind of character allowed: S allows a sign, 9 allows a decimal digit, and other symbols are used for other restrictions. This diagram makes it clear that the storage unit accommodates any value that can be represented as a sign followed by four decimal digits followed by a decimal point followed by two decimal digits. It also makes it clear that the last digit of -31.253 must be dropped before assignment to the storage unit can occur.

A data frame provides useful information about the data type as it is applied to a storage unit. The diagram should be viewed as having three components:

- The characters written in the boxes are the <u>contents</u> of the storage unit. They are the only part of the storage unit that can change; but they do not, by themselves, represent the value of the storage. In the example above, the contents is -003125 and this sequence of characters would not, in itself, be interpreted as the value -31.25.

- The characters in line with the contents but not enclosed in boxes are the <u>interpretation</u> of the storage unit. The contents and the interpretation together are a <u>representation</u> of the value of the storage unit, and the representation is always a valid PL/I constant expression. In the current example, the representation is -0031.25 and this is a valid signed constant and could be used in a program to represent the value -31.25.

- The data frame without contents or interpretation is an indication of the <u>hardware</u> <u>storage</u> required for the storage unit. In the current example, the storage unit has one sign box (one byte in a decimal number) and six decimal-digit boxes (each one byte); therefore the storage unit requires seven bytes of memory.

The data frames are not a part of the PL/I language itself; they are introduced here as a useful way of describing a storage unit. Since a PL/I program cannot depend on the way in which arithmetic values are represented, the purpose of the data frames is <u>not</u> to show how the components of a value can be operated upon.

FIXED DECIMAL STORAGE UNITS

A FIXED DECIMAL storage unit closely approaches everyday notation for a number. The availability of these storage units make it possible to entirely avoid fractional fixed-point binary arithmetic and thus eliminates one of the major problems of computing. Four examples of REAL FIXED DECIMAL storage units follow, each with a different precision attribute. Each example gives the declaration of a variable and then the storage unit that corresponds to the declaration.

```
                                        S 9 9 9 9 9 9   9 9
DCL U1 DEC(8,2);            U1          / / / / / / /./ / /
```

The complete data type for this storage unit is REAL FIXED DECIMAL PRECISION(8,2). The data type accommodates any number with magnitude less than one million to the nearest one hundredth. It would be useful for a sum in dollars and cents and, since it has a sign, it could be used for a credit or a debit.

```
                                          S 9 9 9 9 9 9
DCL U2 DEC(6);                      U2    /_/_/_/_/_/_/.
```

The complete data type for this storage unit is REAL FIXED DECIMAL PRECISION(6,0). It accommodates any integer with magnitude less than one million.

```
                                          S       9 9 9 9 9 9
DCL U3 DEC(6,9);                    U3    /_/.000/_/_/_/_/_/
```

The complete data type for this storage unit is REAL FIXED DECIMAL PRECISION(6,9). The scale-factor is greater than the number-of-digits and therefore filler zeros appear between the decimal point and the first significant digit. The storage unit accommodates a fraction with magnitude less than one thousandth to the nearest billionth.

```
                                          S 9 9 9 9
DCL U4 DEC(4,-2);                   U4    /_/_/_/_/00.
```

The complete data type for this storage unit is REAL FIXED DECIMAL PRECISION(4,-2). The scale-factor is negative and therefore filler zeros appear between the last significant digit and the decimal point. The storage unit accommodates an integer with magnitude less than one million to the nearest hundred.

FIXED BINARY STORAGE UNITS

A FIXED BINARY storage unit can be declared with the same variety of precisions as were just illustrated for the FIXED DECIMAL storage units. In practice, however, a FIXED BINARY storage unit is rarely used with a scale-factor other than zero. Two examples follow.

```
                                          S 1 1 1
DCL V1 FIXED(3);                    V1    /_/_/_/.B
```

The complete data type for this storage unit is REAL FIXED BINARY PRECISION(3,0). The 1 symbol over a box indicates that it must contain a binary digit. The B at the right end shows that the representation is binary. The storage unit can accommodate, for example, the value -2; the representation for that value is -110.B.

```
                                          S 1                 1
DCL V2 FIXED;                       V2    /_/_/--(17)--/_/.B
```

The complete data type is REAL FIXED BINARY PRECISION(17,0). The parenthesized 17 means that there are seventeen digit positions in the storage unit. The storage unit accommodates any integer whose magnitude is less than 2**17 (which is 131072). It is the recommended storage unit for most indexes and subscripts.

FLOAT BINARY STORAGE UNITS

A FLOAT BINARY storage unit is compatible with the floating-point binary hardware operations that are part of many computers. Scientific applications use FLOAT BINARY storage units for physical variables. As already stated in the description of the scale attribute, a FLOAT BINARY storage unit accommodates a value of the form:

m * (2**e)

where m is the mantissa and e is the exponent. The range of a FLOAT BINARY storage unit is determined primarily by the fact that its exponent lies in the range -128 through +127. In GCOS, any FLOAT BINARY storage unit can accommodate values whose magnitudes are in the range 10**-38 to 10***+38. Three examples of FLOAT BINARY storage units are given here.

```
                                      S   1                 1   exp
DCL X1 FLOAT(8);        X1    /_/./_/--(8+)--/_/E/____/B
```

The complete data type for this storage unit is REAL FLOAT BINARY PRECISION(8). The mantissa is a signed fraction with eight or more digits; in GCOS, it has 27 digits. When the representation is written, the exponent is written as an optionally-signed decimal integer, but in the hardware it is represented in seven bits. The B at the end of the representation applies only to the mantissa. The storage unit can accommodate, for example, the value 4.5; the representation used for that value is +.10010000E3B.

```
                                      S   1                 1   exp
DCL X2 FLOAT;           X2    /_/./_/--(27+)--/_/E/____/B
```

The complete data type for this storage unit is REAL FLOAT BINARY PRECISION(27). In GCOS, the mantissa has 27 digits. This data type (without an explicit precision attribute) is used for most scientific variables.

```
                                      S   1                 1   exp
DCL X3 FLOAT(28);       X3    /_/./_/--(28+)--/_/E/____/B
```

The complete data type for this storage unit is REAL FLOAT BINARY PRECISION(28). In GCOS, the mantissa has 63 digits. Thus the increase of the number-of-digits from 27 to 28 causes the change from GCOS single precision floating-point arithmetic to double-precision.

FLOAT DECIMAL STORAGE UNITS

A FLOAT DECIMAL storage unit is used only under exceptional circumstances. It can be used to get more precision than is available from FLOAT BINARY since 59 decimal digits of precision is equivalent to about 196 binary digits of precision. It can also be used to avoid binary representation, but the operations on FLOAT DECIMAL values are so much more expensive than those on FLOAT BINARY values that this course is seldom followed. A FLOAT DECIMAL storage unit accommodates a value of the form:

m * (10**e)

where m is the mantissa and e is the exponent. Any FLOAT DECIMAL storage unit can accommodate values whose magnitudes are in the range 10**127 to 10**-69. One example is given here.

```
                                          S 9          9     exp
DCL Y1 FLOAT DEC(59);          Y1    /  /--(59)--/  /.E/    /
```

The complete storage type for this storage unit is REAL FLOAT DECIMAL PRECISION(59). The mantissa is a signed decimal integer with 59 digits. This storage unit has greater precision and range than any other in PL/I. It occupies 16 words of GCOS memory.

COMPLEX STORAGE UNITS

A COMPLEX storage unit can have any scale, base, or precision attributes. In practice, however, it is always used with the FLOAT BINARY attributes. Since a complex number is a pair of real numbers, a COMPLEX storage unit is formed by placing two REAL storage units together and marking the second as the imaginary part by suffixing an I to it. An example is:

```
DCL Z1 COMPLEX FLOAT;

                 S  1           1   exp   S  1           1   exp
Z1    /  /./  /--(27+)--/  /E/    /B/  /./  /--(27+)--/  /E/    /BI
```

Everything through the first B represents the real part of the storage unit and the remainder is the imaginary part.

Guidelines for Arithmetic Data Types

The choice of data types for the numeric variables is a major part of the design of a program. The first choice that has to be made is between the arithmetic storage types, just described, and the pictured storage types, described later in this section. The arithmetic storage types can be operated on efficiently but require conversions when input/output is performed; they are appropriate when calculations are relatively complicated, as in scientific programming, or when the packing of data is important, as in system programming. The pictured storage types are appropriate when a major consideration is the format of input and output, as in business programming.

Once the general decision has been made, other details must be decided. For an arithmetic storage unit, the mode, scale, base, and precision must be selected. These selections affect the range of the input data accepted by the program, the accuracy of the results developed by the program, the convenience of writing and debugging the program, and the cost of executing the program. The questions of range and accuracy depend on the mathematical analysis of the algorithm being programmed. The questions of convenience and cost depend on the human and mechanical factors of the programming system. Guidelines for the choice of the attributes of an arithmetic storage unit are given here.

THE CHOICE OF MODE

Except for scientific applications, the mode is always REAL; and since the default mode is REAL, most programs never make explicit use of a mode attribute. Within a given scientific program, the requirement for a COMPLEX mode attribute is determined by the mathematical formulation of the given program. The choice of mode is entirely a question of range.

THE CHOICE OF SCALE AND BASE

The choice of the scale attribute is a choice between the efficiency of the fixed-point operations and the wide range of floating-point arithmetic. The choice of the base attribute is a choice between the efficiency of binary operations and the convenient familiarity of decimal representation. The following rules contribute to the selection of the scale and base for a particular storage unit:

1. When a storage unit is used for an exact integer that remains in a reasonable range, it should be FIXED BINARY. This rule applies to subscripts, indexes, and counters.

2. When a storage unit is used for a non-integer quantity that must be approximated with care, it should be FIXED DECIMAL. An example is a dollars-and-cents quantity.

3. When a storage unit is used for a quantity that is inherently approximate, it should be FLOAT BINARY; however, if an exceptional requirement for precision exists, the storage unit should be FLOAT DECIMAL. Most physical quantities, such as weight, length, speed, and so on, should have the FLOAT attribute.

In some business applications, it is convenient to use the DECIMAL base for all storage units and thus avoid the mixing of DECIMAL and BINARY values; this policy requires a departure from the first of the rules just given.

THE CHOICE OF PRECISION

The precision attribute gives the number-of-digits for every arithmetic storage unit and gives the scale-factor for FIXED storage units. The choice of precision for FLOAT variables is usually easy because there is no danger of overflow and a detailed analysis of the accuracy of the calculation is impossible. The choice of precision for a FIXED variable is much more difficult because of the danger of overflow and the possibility that significant digits will be lost. Some form of analysis is usually required for the choice of precision for a FIXED variable.

The defaults for the number-of-digits depend on a given implementation of PL/I and can vary from one computer to another. Consider a FIXED variable that is used as a counter. If an analysis has shown that the variable will never require more than 10 binary digits, then it should be declared FIXED(10). If, on the other hand, a less precise analysis has determined that a "fairly large" capacity will suffice, then it should be declared FIXED. In the first case, the programmer makes a specific assertion about the use of the variable; in the second case, the programmer gives the compiler latitude to select an efficient representation for the given hardware.

When a large counter is required, the number of digits should be kept less than 35 if possible. Although a FIXED(35) variable occupies only one word of GCOS memory, operations on the variable tend to get into double precision. Thus FIXED(30) would be a better choice.

ORDINARY STRING STORAGE

A string storage unit is either ordinary or pictured. The differences between the two kinds of string storage are more important than their similarities, so they are described separately. Ordinary string storage, described here, is primarily used for text of a general nature, such as column headings, error messages, or complete documents. Pictured string storage, described later, is primarily used for numeric values that are represented as character strings.

Ordinary String Data Types

The complete data type for an ordinary string storage unit is given by a sequence of two string attributes. These attributes are the string-type and the variability, as described in the following paragraphs.

THE STRING-TYPE ATTRIBUTE

The string-type attribute has one of the following forms:

CHARACTER(ml)

BIT(ml)

where ml is the maximum-length of the string. The maximum length must be an extent: that is, it must have the form:

exp

exp REFER ref

*

where exp is an expression and ref is a reference. The first form is used in most cases; it must yield a value that can be converted to a FIXED BINARY(24) value. The second two forms are described in later sections on "Storage Management" and "Procedure Invocation", respectively.

A storage unit declared CHARACTER(ml) can accommodate a sequence of n ASCII characters, where n is the value of ml at the time the storage unit is allocated. Similarly, a storage unit declared BIT(m1) can accommodate a sequence of n bits, where n is the value of ml at the time the storage unit is allocated. A bit is one of the characters 0 or 1.

THE VARIABILITY ATTRIBUTE

The variability attribute is one of the following keywords:

NONVARYING
VARYING

An ordinary string storage unit with the NONVARYING attribute can accommodate a string of only one length: the maximum length specified in the string-type attribute. When a shorter string is assigned to a NONVARYING storage unit, blanks or zeros (for a character or bit string, respectively) are added to the right end of the string until it has the required maximum length. An ordinary string unit with the VARYING attribute can accommodate a string of any length from zero up to and including the maximum length given in the string-type attribute.

ABBREVIATIONS AND DEFAULTS

The following abbreviations are accepted for the keywords used in ordinary string attributes:

Keyword	Abbreviation
CHARACTER	CHAR
VARYING	VAR

The defaults for the string attributes are:

Omitted Item	Default
variability attribute	NONVARYING
maximum-length	1

Some examples of the application of these rules are:

Complete Data Type	Recommended Form
CHARACTER(N+1) VARYING	CHAR(N+1) VAR
CHARACTER(1) NONVARYING	CHAR
BIT(1) NONVARYING	BIT

The last case is especially useful because a one-bit string is used as a Boolean value in PL/I.

Examples of Ordinary String Storage

As an example of the use of an ordinary string storage unit, consider the following program:

```
P:  PROC;
        DCL BETA CHAR(15);
        ...
        BETA = "JOHN Q. SMITH";
        ...
        END;
```

The storage unit for BETA can be diagrammed as:

```
          CHAR(15)
BETA  / "JOHN Q. SMITH  " /
```

Because the variable is NONVARYING by default, two blanks are added at the right end of the assigned character string to make it 15 characters long.

A data frame can be used for an ordinary string storage unit. For the variable BETA, just discussed, the resulting diagram is:

```
                X X X X X X X X X X X X X X X
BETA  "/J/O/H/N/ /Q/./ /S/M/I/T/H/ / /"
```

The diagram has 15 boxes, one for each character in the stored string. The X over each box indicates that the box can hold any ASCII character.

CHARACTER STORAGE UNITS

Two examples of CHARACTER storage units are given here; one is NONVARYING and the other is VARYING.

```
DCL S1 CHAR(80);          S1   "/ /--(80)--/ /"
                                  X       X
```

The complete data type for this storage unit is CHARACTER(80) NONVARYING. The storage unit accommodates a character string that is exactly eighty characters long; it could be used, for example, to store the contents of an eighty-column card.

```
DCL S2 CHAR(6) VAR;       S2   / 3 /"/S/A/M/&/W/2/"
                                 CNT  X X X X X X
```

The complete data type is CHARACTER(6) VARYING. The storage unit can accommodate any character string that has from zero to six characters. The storage unit is shown with a value in it; that value is "SAM". The CNT box gives the current length of the string in the storage unit; the current length is filled in each time a new value is assigned to the storage unit. The last three character positions in the storage unit contain characters which, presumably, are left from previous values of the storage unit.

BIT STORAGE UNITS

The BIT storage units are defined in the same general way as the CHARACTER storage units. In practice, however, BIT storage units are used in rather specialized ways. Two examples are given here.

```
                                                    1
DCL T1 BIT;                          T1    "/_/"B
```

The complete data type is BIT(1) NONVARYING. The storage type is used when a Boolean variable is required. It can have only two values, and these are interpreted as follows:

"1"B means "true"

"0"B means "false"

Such values are used in IF statements, loops, and other statements that control program flow.

```
                                                  1 1 1 1 1
DCL T2 BIT(5);                       T2    "/_/_/_/_/_/"
```

The complete data type is BIT(5) NONVARYING. The storage type could be used to hold five flags, each of which represented the truth or falsity of some position. Such bit strings are especially useful in programs for process control.

Guidelines for Ordinary String Data Types

A CHARACTER storage unit can be used in a general way: it is useful for holding a single character, for holding a proper name or a short phrase, or for holding a document of considerable size. Further, both the VARYING and the NONVARYING CHARACTER storage unit are useful. In contrast, a BIT storage unit is specialized: it is usually just one BIT in length and it is almost always NONVARYING.

As the diagrams given in the examples suggest, the amount of storage required for a string depends on the maximum-length, not on the current length. Thus a CHARACTER(1000) VARYING storage unit occupies the same amount of storage (251 words) regardless of whether it contains a string value of length one or 1000.

A NONVARYING string is handled more efficiently than a VARYING string. Specifically, a VARYING string requires an extra GCOS word to keep the current-length counter in and the code for accessing the storage unit is more complicated. Therefore, when efficiency is the only consideration, a NONVARYING string should be used.

Often the choice of the variability attribute is determined by the nature of the data. For example, consider the storage for a textbook that is being edited. An individual line might be kept in a NONVARYING string storage unit, since lines are of constant length. On the other hand, an individual word drawn from the text might be stored in a VARYING storage unit.

PICTURED STRING STORAGE

Pictured-string storage units offer an almost complete alternative to arithmetic and ordinary string storage units; that is, many useful programs are written in a natural and convenient way using only pictured storage units. Pictured storage is most often used in business applications, but it can be used very effectively elsewhere.

The pictured storage unit eliminates the use of a special representation for numbers inside the computer; it uses the same representation inside that people use on the outside on cards and listings. The representation is a sequence of characters arranged in a special way: decimal digits, a decimal point, a properly placed sign, and so on. When pictured storage is used, input/output does not involve a conversion between an encoded internal representation and the external text; instead, it simply requires the transmission of a character string.

A pictured storage unit is thought of as having two values. When it is referenced in a context that requires a character string (as in an output statement), then its value is a character string. When it is referenced in a context that requires an arithmetic value (as in a calculation), then its value is an arithmetic value.

The use of a character string to store a numeric value is less efficient than the use of a specially encoded sequence of bits. However, some efficiency can be achieved if the exact form of the character string in a given storage unit is known to the compiler. The purpose of the "picture" that is given in the declaration of a pictured storage unit is to establish the exact form of the string accommodated by the storage unit. As an example, consider the following declaration:

```
DCL OMEGA PICTURE"S999";
```

This declaration asserts that the storage unit designated by OMEGA will accommodate a string of four characters. It also specifies that the first character will be a sign and the remaining characters will be decimal digits. This allows the compiler to conclude that the value of OMEGA can always be interpreted either as a character-string value of data type CHAR(4) or as an arithmetic value of data type REAL FIXED DECIMAL(3).

Pictured Data Types

The complete data type for a pictured storage unit is a picture attribute followed by an optional mode attribute. These attributes are described in the following paragraphs.

THE PICTURE ATTRIBUTE

The picture attribute has the form:

PICTURE"p"

where p is the picture. The picture is a sequence of indicators, each optionally preceded by a replicator. A replicator is a parenthesized, unsigned, decimal integer constant. An example of a picture attribute is:

PICTURE"(4)9V.99DB"

In this example, the picture is:

(4)9V.99DB

The replicator (4) means that the following indicator is treated as if it appeared four times. Thus the equivalent picture attribute is:

PICTURE"$$$$$9V.99DB"

The picture contains five different indicators, as follows:

$ 9 V . DB

Each indicator has a special interpretation, and the order in which the indicators are given is significant. A full description of the permitted pictures and their interpretations is given a little later.

The Classification of Indicators

A complete list of the indicators follows. The indicators are classified under functional headings, and these same headings are used later in this section when the individual indicators are fully defined.

Classification	Indicators
no-suppression digit indicator	9
decimal-point indicator	V.
sign indicators	S + - CR DB
dollar indicator	$
zero-suppression digit indicators	Z * Y
drifting-sign digit indicators	S + -
drifting-dollar digit indicator	$
insertion-character indicator	. , / B
arithmetic decimal-point indicator	V
fixed-point scale-factor indicator	F(n)
floating-point indicators	E K
floating-point scale-factor indicator	F(n)
non-numeric indicators	9 A X

In the scale-factor indicators, n is an optionally-signed decimal integer constant. The following distinctions apply:

- The 9 indicator is a no-suppression digit indicator if it appears in a numeric picture; otherwise, it is a non-numeric indicator and the interpretation is slightly different.

- The S indicator is a sign indicator if it is the first S in a fixed-point picture, a mantissa picture, or an exponent picture; otherwise, it is a drifting-sign digit indicator. Parallel considerations apply to the classification of +, -, and $.

- The F(n) indicator is a floating-point scale-factor indicator if it appears in a picture with a floating-point indicator; otherwise, it is a fixed-point scale-factor indicator.

Observe that eight of the indicators are digit indicators, as follows:

9 Z * Y S + - $

where some of these are digit indicators only in certain contexts, as just noted.

The Classification of Pictures

It is useful to classify pictures in the following way:

- A picture is non-numeric if it contains either of the following indicators:

  A X

  Otherwise, the picture is numeric.

- A numeric picture is floating-point if it contains either of the following indicators:

  E K

  Otherwise, the picture is fixed-point.

Examples are:

Picture Attribute	Classification
"99A999"	non-numeric
"999999"	numeric
"$9V.99CR"	numeric
"S9V.999999ES999"	floating-point numeric
"SV.999999KS99"	floating-point numeric
"S9V.999999"	fixed-point numeric

THE MODE ATTRIBUTE

The mode attribute is one of the following keywords:

REAL
COMPLEX

This attribute is the same as the mode attribute for arithmetic storage units. It is used only with a picture attribute that is numeric (that is, that does not have an A or X indicator). When the REAL attribute is used, the picture remains as it is given; when the COMPLEX attribute is used, the picture is doubled to provide for the real and imaginary parts of a complex value. For example, the data type:

```
PICTURE"S999" COMPLEX
```

describes a storage unit that has eight character positions, the first four of which accommodate a signed, three-digit value for the real part and the second four of which accommodate a signed, three-digit value for the complex part.

ABBREVIATIONS AND DEFAULTS

The following abbreviations are accepted for the keywords used in the pictured string attributes:

Keyword	Abbreviation
PICTURE	PIC
COMPLEX	CPLX (not recommended)

A default convention for the numeric pictured string data type is:

Omitted Item	Default
mode attribute	REAL

Interpreting Pictured Storage

The interpretation of a storage unit determines the way in which a value is assigned to the storage unit and later fetched. The operations of assignment and fetch do not include data type conversions that are necessary to adjust the value of the storage unit to its context; the conversions are separate operations, discussed in a later section, under "Value Conversion". The operations of assignment and fetch are very simple for arithmetic storage units and ordinary string storage units, and require little comment. However, a pictured storage unit has two interpretations, depending on the context in which it is used: it can be interpreted as a character-string storage unit or an arithmetic storage unit. Complications arise in keeping these two interpretations consistent with one another.

THE CHARACTER-STRING INTERPRETATION OF PICTURED STORAGE

The character-string interpretation of a pictured storage unit requires a definition of a related character-string data type. Once the related type is determined, the arithmetic operations for assignment and fetching can be defined.

Related Character Types

The related character type for a pictured storage unit is defined as:

CHARACTER(*i*) NONVARYING

where *i* is obtained by counting all the characters in the picture except for those that appear in the following indicators:

V	the arithmetic decimal-point indicator
K	the non-printing floating-point indicator
F(*n*)	the scale-factor indicator

These indicators are defined later; they are not counted because they contribute only to the arithmetic interpretation of a pictured storage unit.

As an example of the determination of the related character type, consider the picture attribute:

PIC"S999V.99F(-12)"

This picture attribute has the related character type:

CHARACTER(7) NONVARYING

The length 7 is determined by counting the four characters S999, ignoring the indicator V, counting the three characters .99, and ignoring the indicator F(-12).

The following is an example of a non-numeric picture attribute:

PIC"AAXX9"

This picture attribute has the related character type:

CHARACTER(5) NONVARYING

The length 5 is determined by counting all the characters between the quote characters.

The data frame diagram for a pictured storage unit closely resembles that for a nonvarying character string. For the first example of a picture attribute just given, the data frame is:

```
          S 9 9 9 V.9 9 F(-12)
PIC "/_/_/_/_/_/_/_/_/"
```

For the second example of a picture attribute the data frame is:

```
          A A X X 9
PIC "/_/_/_/_/_/"
```

Observe that each data frame has a character position for each indicator that is counted in the determination of the related character type for the picture.

Character-String Assignments

The character-string assignment operation occurs when any value is assigned to a non-numeric pictured storage unit. Before the character-string assignment operation begins, the given.value is converted to the related character-string data type for the pictured storage unit. The conversion operation is not part of the assignment operation, and is covered later, in the section on "Value Conversion".

Because of the way the related character-string data type is determined, the converted character-string value has exactly the number of characters that is required by the picture. Each character in the converted character-string value is checked against the corresponding indicator. If a character does not satisfy the requirements of the indicator, the CONVERSION condition occurs. The CONVERSION condition is described later, in the section on "Value Conversion".

As an example of the assignment of a character-string value to a pictured storage unit, suppose the variable X has been declared as follows:

```
DCL X PIC"XX9999";
```

and suppose the following assignment statement is executed:

```
X = "0023.15";
```

The related character-string data type for the given picture is:

```
CHAR(7)
```

The character-string constant in the assignment statement already has exactly this data type, so no preliminary conversion is necessary. The assignment operation can begin. The picture is examined from left to right, and each indicator is checked against the corresponding character. The CONVERSION condition occurs on the fifth indicator; it is a 9, which specifies a digit, and the corresponding character is a period. Therefore the assignment cannot be made.

Character-String Fetches

The character-string fetch operation occurs when the value of a numeric pictured storage unit is fetched in a context that requires a character-string value or when the value of a non-numeric pictured storage unit is fetched in any context.

The character-string fetch operation provides a value whose data type is the related character-string data type for the picture. It can never be invalid or cause a condition to occur.

THE ARITHMETIC INTERPRETATION OF PICTURED STORAGE

The arithmetic interpretation of a pictured storage unit requires the definition of a related arithmetic data type. Once the related type is determined, the arithmetic operations for assignment and fetching can be defined.

Related Arithmetic Data Types

The related arithmetic data type for a given picture attribute has one of the forms:

FIXED DECIMAL(p,q)

FLOAT DECIMAL(p)

where p and q are determined from the given picture attribute. The first form is used for a fixed-point picture and the second, for a floating-point picture.

For a fixed-point picture, the related precision attribute is determined as follows:

p is the number of digit indicators in the entire picture.

q is the number of digit indicators to the right of the V indicator minus the value given by the scale-factor indicator. If there is no V indicator in the picture, it is assumed to be at the right end of the picture. If there is no scale-factor indicator in the picture, it is assumed to be F(0).

As an example, consider the picture attribute:

PIC"S999V.99F(-2)"

This picture has the related arithmetic data type:

FIXED DECIMAL(5,4)

The value p=5 was obtained by ignoring the S indicator, counting the three 9 indicators, ignoring the V. indicators, counting the two 9 indicators and ignoring the scale-factor indicator. The value q=4 was obtained by counting the two 9 indicators after the V indicator and then subtracting the value given by the scale-factor indicator, -2.

For a floating-point picture, the related precision attribute is determined as follows:

p is the number of digit indicators in the mantissa of the picture; that is, before the E or K indicator.

As an example, consider the picture attribute:

```
PIC"S9V.9999ES99"
```

This picture has the related arithmetic data type:

```
FLOAT DECIMAL(5)
```

The value p=5 was obtained by ignoring the S indicator, counting the 9 indicator, ignoring the V. indicator, counting the four 9 indicators, and ignoring the E indicator and everything after it.

Arithmetic Assignments

The arithmetic assignment operation occurs when an arithmetic or bit-string value is assigned to a numeric pictured storage unit. Before the arithmetic assignment operation begins, the given value is converted to the related arithmetic data type for the pictured storage unit. During the conversion, a condition may occur to report that the assigned value is out of range; also, the rightmost digits may be truncated or rounded off if the related arithmetic data type cannot accommodate them. The conversion operation is not part of the assignment operation, and is covered later, in the section on "Value Conversion".

Because of the way the related arithmetic data type is determined, the converted arithmetic value has exactly the number of digits that are required by the picture. Other parts of the pictured value are determined by the indicators in the picture, in accordance with the detailed definitions given later in this section. The result of the assignment operation is a character sequence that is placed in the pictured storage unit.

A simple example of the assignment of an arithmetic value to a pictured storage unit follows. Suppose the variable X has been declared as follows:

```
DCL X PIC"$99S";
```

and suppose the following assignment statement is executed:

```
X = 3;
```

The related arithmetic data type for the given picture is:

```
DEC(2)
```

Before the assignment operation can begin, the given value must be converted to the related arithmetic data type; the result is:

```
+03.
```

The value is shown in the representation that is used to store a DEC(2) value. Now the assignment operation can begin. The picture is examined from left to right, and each indicator is processed as follows:

1. The first indicator is $. This indicator means that a dollar sign must appear at this point in the character string. Therefore, the first character is $.

2. The second indicator is 9. This means that a decimal digit must appear, and the first digit of the converted given value is used. Therefore the second character is 0.

3. The third indicator is 9. This means, again, that a decimal digit must appear, and the second digit of the converted given value is used. Therefore the third character is 3.

4. The fourth indicator is S. This means that the sign must appear, and the sign of the converted given value is used. Therefore the fourth character is +.

The resulting character sequence is:

```
$03+
```

and this is the value that is placed in the pictured storage unit named X.

Arithmetic Fetches

The arithmetic fetch operation occurs when the value of a numeric pictured storage unit is fetched in a context that requires an arithmetic or bit-string value. After the fetch is complete, some conversion of the arithmetic result may be required; but this conversion is not part of the fetch operation.

The arithmetic fetch operation is the inverse of the assignment operation. That is, if a given arithmetic value is assigned to a pictured storage unit, then a subsequent fetch operation will obtain exactly the value that was assigned.

A simple example of the fetching of an arithmetic value from a pictured storage unit follows. The example is parallel to the example just given, in the discussion of assignment. Suppose the variable X has been declared as follows:

```
DCL X PIC"$99S";
```

and suppose that X appears in a context that requires an arithmetic value, such as the expression:

```
X+1
```

The related data type for the given picture is:

```
DEC(2)
```

Let the current value of X be the character sequence:

```
$03+
```

In order to fetch the arithmetic value of the storage unit, the picture is examined from left to right, and each indicator is interpreted as follows:

1. The first indicator is $. This indicator makes no contribution to the arithmetic interpretation and is ignored.

2. The second indicator is 9. This indicator contributes a digit to the arithmetic interpretation. Therefore, the first digit of the arithmetic interpretation is 0.

3. The third indicator is 9. This indicator also contributes a digit to the arithmetic interpretation. Therefore, the second digit of the arithmetic interpretation is 3.

4. The fourth indicator is S. This indicator contributes the sign for the arithmetic interpretation. Therefore, the sign of the arithmetic interpretation is +.

The resulting arithmetic value is:

+03.

which is the correct representation for a DEC(2) value.

Fixed-Point Pictures

There are many kinds of fixed-point pictures. However, any fixed-point picture can be constructed in three steps, as follows:

1. Start with a sequence of one or more 9 indicators. (The 9 indicator is called the no-suppression digit indicator, and means that a digit must appear in the corresponding position of the character-string value.)

2. Optionally, insert any decimal-point, sign, dollar, or insertion-character indicators. Certain restrictions on the way these indicators are used must be observed; for example, a dollar indicator must be at the beginning or end of the sequence, not between two 9 indicators.

3. Optionally, choose a zero-suppression, drifting-sign, or drifting-dollar indicator and, proceeding from left to right, replace one or more of the 9 indicators in the picture. Restrictions on the choice of the new indicator must be observed; for example, a drifting-dollar indicator can be used only if the leftmost 9 is immediately preceded by a dollar indicator.

These three steps could be carried out with pencil and paper as an exercise; however, they are presented here as a convenient form of definition for the fixed-point pictures. The various restrictions mentioned in Steps 2 and 3 are given later, in the descriptions of each of the indicators.

An example of the construction of a picture according to the steps just given is:

```
9999
$99V.99
$$$V.99
```

Here, a sequence of four 9 indicators was created by Step 1; than a dollar indicator and a decimal-point indicator were inserted by Step 2; and finally a drifting-dollar digit indicator was chosen to replace some leading 9 indicators (two of them). A second example is:

```
999
999
ZZZ
```

Here, the optional Step 2 was skipped and the Z zero-suppression indicator was chosen to replace some leftmost 9 indicators (all three of them).

Once a valid fixed-point picture has been constructed, its interpretation must be determined; that is, the assignment and fetch operations must be defined. Most of the interpretation can be expressed in terms of the individual indicators that appear in the picture. However, there is one rule that pertains to the picture as a whole:

> If a zero value is assigned to a fixed-point picture that does not contain a 9 indicator, then the entire character string becomes a sequence of ␣ (blank) characters (if the suppression was not by a * indicator) or a sequence of * characters (if suppression was by the * indicator).

For example, if the picture attribute is:

```
PIC"$ZZV.ZZS"
```

and the value is zero, then the character-string value is <u>not</u>

```
"$␣␣.␣␣+"
```

which is the result of suppressing zeros, but rather is

```
"␣␣␣␣␣␣␣"
```

which is the result of setting <u>all</u> characters to blanks.

THE NO-SUPPRESSION DIGIT INDICATOR

The most basic of all the indicators is the <u>no-suppression digit indicator</u>, which is defined as follows:

> 9 means that a decimal digit must appear in the corresponding position of the character-string value. In the arithmetic interpretation, the corresponding digit is interpreted as a part of a decimal number.

The name "no-suppression digit indicator" means that this indicator specifies a digit that is never suppressed (replaced by a blank when it is zero).

An example of a pictured attribute that uses the no-suppression digit indicator is:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"999"	180	CHAR(3)	"180"	DEC(3)	+180.
	2		"002"		+002.
	13.99		"013"		+013.
	1000		(undefined)		(undefined)
	-22		(undefined)		(undefined)

For this set of examples, it is assumed that some variable, say X, has been declared with the given picture attribute:

```
DCL X PIC"999";
```

For each line of the example, it is assumed that a value has been assigned to X. For the first line, the assignment is:

```
X = 180;
```

Each line gives the data type and value for two contexts. The entry under "Char Type and Value" applies when the variable is referenced in a context that requires a character-string value. The entry under "Arith Type and Value" applies for any other context; that is, when the variable is referenced in a context that requires an arithmetic or bit-string value.

The assignment of 2 to the variable shows that leading zeros are preserved in the character string value when 9 indicators are used. The last three examples illustrate three ways in which a value can be arithmetically modified by assignment to a pictured storage unit:

- When the assigned value has more accuracy than provided for by the picture (as in the case of 13.99), the low-order digits are dropped without rounding. Although the absence of rounding is not always acceptable, this value modification is thought of as an approximation rather than an error; and no exceptional condition occurs when such an assignment is performed.

- When the assigned value is too big for the picture (as in the case of 1000), the value assigned to the storage unit is undefined. This value modification is, of course, an error and the SIZE condition occurs. Usually a SIZE condition terminates program execution, but recovery from the error is described in the section on "Condition Handling".

- When the assigned value is negative and the picture does not have a sign indicator (as in the case of -22), then the value assigned to the storage unit is undefined, and the SIZE condition occurs to indicate an error.

THE DECIMAL-POINT INDICATOR

The decimal-point indicator is made up of two characters, and is defined as follows:

V. means that if there is no scale-factor in the picture then the decimal point appears at this position, both in the character-string and the arithmetic value of the pictured storage unit.

The V. indicator can appear no more than once and, except for intervening insertion-character indicators (described later), the indicator must be adjacent to a digit indicator. For the rare case that there is a scale-factor in the picture, see the definition of the arithmetic decimal-point indicator, given later.

An example of a picture that makes use of a decimal-point indicator is:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"999V.99"	500.98	CHAR(6)	"500.98"	DEC(5,2)	+500.98
	16		"016.00"		+016.00
	0		"000.00"		+000.00
	6.839		"006.83"		+006.83
	1500.98		(undefined)		(undefined)
	-1		(undefined)		(undefined)

The last three assignments illustrate again the three kinds of arithmetic value modifications mentioned under "The No-Suppression Digit Indicator". Replicators could have been used in the picture in any of the following ways:

PIC"(3)9V.(2)9" PIC"(3)9V.99" PIC"999V.(2)9"

A second example of the use of the decimal-point indicator is:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"999V."	180	CHAR(4)	"180."	DEC(5)	+180

Here the effect of V. is to make the decimal point in the character string explicit; it could be omitted without changing the arithmetic value. Compare this example to the first line in the examples for "The No-Suppression Digit Indicator".

Each of the two characters that make up the decimal-point indicator is, itself, an indicator. The V is the arithmetic decimal-point indicator and the . is one of the insertion-character indicators. The separate use of V and . is rare, but their definitions are included, for completeness, later in this section.

THE SIGN INDICATORS

A picture can contain a sign indicator. The sign indicators are defined as follows:

S means that + or - must appear in the corresponding character position. In the arithmetic interpretation, the characters + and - mean the value is positive or negative, respectively.

\+ means that a + or a blank must appear in the corresponding character position. In the arithmetic interpretation, the characters + or blank mean the value is positive or negative, respectively.

\- means that a blank or a - must appear in the corresponding character position. In the arithmetic interpretation, the characters blank or - mean the value is positive or negative, respectively.

CR means that two blanks or CR must appear in the corresponding two character positions. For the arithmetic interpretation, the two blanks or CR mean the value is positive or negative, respectively.

DB means that two blanks or DB must appear in the corresponding two character positions. For the arithmetic interpretation, the two blanks or DB mean the value is positive or negative, respectively.

Only one sign indicator can occur in a fixed-point picture. Any one of the indicators

S + -

can occur before the first digit indicator in the picture; this leading position is in accord with the common usage of signs. Any one of the indicators

S + - CR DB

can occur after the last digit indicator; this is in accord with some business usage.

Examples of the use of a sign indicator at the beginning of a picture are:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"S999"	82	CHAR(4)	"+082"	DEC(3)	+082.
	0		"+000"		+000.
	-82		"-082"		-082.
PIC"+999"	82	CHAR(4)	"+082"	DEC(3)	+082.
	0		"+000"		+000.
	-82		"b082"		-082.
PIC"-999"	82	CHAR(4)	"b082"	DEC(3)	+082.
	0		"b000"		+000.
	-82		"-082"		-082.

Two of these assignments are of particular interest. The assignment of a negative value to a variable with a + sign indicator gives a blank for the sign (second picture, third assignment); this is not a common handling of the sign, and the use of the + indicator is not recommended. In contrast, the assignment of a number to a variable with the - indicator (third picture) follows the familiar conventions for the sign, and the use of the - is a useful alternative to the S indicator.

Examples of the use of a sign indicator at the end of a picture are:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"99V.99S"	5	CHAR(6)	"05.00+"	DEC(4,2)	+05.00
	-5		"05.00-"		-05.00
PIC"99V.99+"	5	CHAR(6)	"05.00+"	DEC(4,2)	+05.00
	-5		"05.00␣"		-05.00
PIC"99V.99-"	5	CHAR(6)	"05.00␣"	DEC(4,2)	+05.00
	-5		"05.00-"		-05.00
PIC"99V.99CR"	5	CHAR(7)	"05.00␣␣"	DEC(4,2)	+05.00
	-5		"05.00CR"		-05.00
PIC"99V.99DB"	5	CHAR(7)	"05.00␣␣"	DEC(4,2)	+05.00
	-5		"05.00DB"		-05.00

As before, the + indicator behaves contrary to familiar conventions. The CR (credit) and DB (debit) indicators are designed especially for business programming and behave in accordance with accounting conventions.

THE DOLLAR INDICATOR

A picture can contain a dollar indicator, which is defined as follows:

$ means that a $ must appear in the corresponding position of the character-string value. In the arithmetic interpretation, the indicator is ignored.

A dollar indicator can occur in either of two ways in a fixed-point picture. First, it can appear anywhere before the first digit position. Second, it can occur anywhere after the last digit position and before a CR, DB, S, +, or - indicator (if there is one).

Examples of the use of the dollar indicator are:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"$999V.99"	20	CHAR(7)	"$020.00"	DEC(5,2)	+020.00
PIC"999V.$DB	-478	CHAR(7)	"478.$DB"	DEC(3)	-478.

THE ZERO-SUPPRESSION DIGIT INDICATORS

A fixed-point picture can contain zero-suppression digit indicators, which are defined as follows:

Z means that a decimal digit or a blank occurs in the corresponding position of the string value. The blank occurs if the character would be a leading zero; otherwise, a decimal digit occurs. In the arithmetic interpretation, a blank that corresponds to the indicator is interpreted as a zero.

* means that a decimal digit or a * occurs in the corresponding position. The definition is parallel to that for Z.

Y means that a decimal digit or a blank occurs in the corresponding position of the string value. The blank occurs if the corresponding character would be zero (regardless of whether or not it would be a leading zero). In the arithmetic interpretation, a blank that corresponds to the indicator is interpreted as a zero.

A leading zero is a digit that is a zero, is not preceded by a nonzero digit, and (less obviously) is to the left of a decimal-point indicator, V. or V, if a decimal-point indicator appears.

Each of these zero-suppression digit indicators can be used as an alternative to a no-suppression digit indicator, 9, in a fixed-point picture. The first two indicators are subject to the following restrictions:

- The Z indicators in a picture must not be preceded by any other kind of digit indicator; similarly, the * indicators must not be preceded by any other kind of digit indicator.
- If a Z indicator is used to the right of a V. or V indicator, then all digit indicators in the fixed-point picture must be Z indicators. Similarly, if a * is used to the right of a V. or V, then all digit indicators must be * indicators.

These restrictions can be illustrated by considering the following picture:

```
PIC"S99V.99"
```

The only valid pictures that have the same pattern of digit indicators but use Z or * are:

```
PIC"SZ9V.99"          PIC"S*9V.99"
PIC"SZZV.99"          PIC"S**V.99"
PIC"SZZV.ZZ"          PIC"S**V.**"
```

Examples of the suppression of leading zeros by means of the Z indicator are:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"-ZZV.ZZ"	-12.18	CHAR(6)	"-12.18"	DEC(4,2)	-12.18
	-2.18		"-␣2.18"		-02.18
	-0.18		"-␣␣.18"		-00.18
	-0.08		"-␣␣.08"		-00.08
	-0.00		"␣␣␣␣␣␣"		+00.00
PIC"-Z9V.99"	-0.18	CHAR(6)	"-␣0.18"	DEC(4,2)	-00.18
	-0.00		"+␣0.00"		+00.00

Observe that, in the last assignment for the first picture, all of the digits are suppressed and therefore the entire string value is blank.

Examples of the suppression of leading zeros by means of the * indicator are:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"$**V.**"	56.20	CHAR(6)	"$56.20"	DEC(4,2)	+56.20
	0.03		"$**.03"		+00.03
	0		"******"		+00.00

Examples of the suppression of zeros by means of the Y indicator are:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"99Y99"	44444	CHAR(5)	"44444"	DEC(5)	+44444.
	44044		"44␣44"		+44044.
	5		"00␣05"		+00005.
PIC"YYY99"	10000	CHAR(5)	"1␣␣00"	DEC(5)	+10000.

THE DRIFTING-SIGN DIGIT INDICATORS

A fixed-point picture can contain drifting-sign digit indicators, which are defined as follows:

S has the same meaning as Z, except that it indicates that the sign (+ or -) must be moved to the right over a suppressed leading zero.

\+ has the same meaning as Z, except that it indicates that the sign (+ or blank) must be moved to the right over a suppressed leading zero.

\- has the same meaning as Z, except that it indicates that the sign (blank or -) must be moved to the right over a suppressed leading zero.

When a sequence of S indicators appears in a picture, the leftmost one is interpreted as a sign indicator and the remaining ones are interpreted as drifting-sign digit indicators. The same interpretation is applied to a sequence of + or - indicators. Drifting-sign digit indicators are subject to the same restrictions as the Z zero-suppression indicator.

Some examples of the use of drifting-sign digit indicators are:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"SSS9V.99"	-180.39	CHAR(7)	"-180.39"	DEC(5,2)	-180.39
	-.39		"ØØ-0.39"		-000.39
PIC"----V.--"	-180.39	CHAR(7)	"-180.39"	DEC(5,2)	-180.39
	-.39		"ØØØ-.39"		-000.39
	-.09		"ØØØ-.09"		-000.09
	-0		"ØØØØØØØ"		+000.00

The two attributes just given are similar to the following:

PIC"SZZ9V.99"

PIC"-ZZZV.ZZ"

The only difference is in the placement of the character that represents the sign. When drifting-sign digit indicators are used, the sign character occupies the position of the rightmost suppressed leading zero. When the Z indicator is used, the sign character remains at the position specified by the sign indicator and (in these examples) is the first character.

THE DRIFTING-DOLLAR DIGIT INDICATOR

A fixed-point picture can contain drifting-dollar digit indicators, defined as follows:

$ has the same meaning as Z, except that it indicates that the dollar character must be moved to the right over a suppressed leading zero.

When a sequence of $ indicators appears in a picture, the leftmost one is interpreted as a dollar indicator and the remaining ones are interpreted as drifting-dollar digit indicators. Drifting-dollar digit indicators are subject to the same restrictions as the Z zero-suppression indicators.

Some examples of the use of drifting-dollar digit indicators are:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"$$$9V.99"	200	CHAR(7)	"$200.00"	DEC(5,2)	+200.00
	.03		"ØØ$0.03"		+000.03
	0		"ØØ$0.00"		+000.00
PIC"$$$$V.$$"	200	CHAR(7)	"$200.00"	DEC(5,2)	+200.00
	.03		"ØØØ$.03"		+000.03
	0		"ØØØØØØØ"		+000.00

THE INSERTION-CHARACTER INDICATORS

A picture can contain insertion-character indicators, which are defined as follows:

. means that a period must appear in the corresponding character position unless zero suppression applies.

, means that a comma must appear in the corresponding character position unless zero suppression applies.

/ means that a slash must appear in the corresponding character position unless zero suppression applies.

B means that a blank must appear in the corresponding character position unless zero suppression applies. Observe that the indicator is the letter B but it inserts a blank character.

In the arithmetic interpretation of a pictured storage unit, all insertion characters are ignored. There is no restriction on the way insertion characters are placed in a picture.

The definitions just given refer to the possibility that zero suppression applies to an insertion character, and that possibility is now described. An insertion character is suppressed when it immediately follows a suppressed digit or another suppressed insertion character; however, no suppression occurs to the right of a V indicator. The effect of suppression is the same as for a digit; that is, the insertion character is replaced by a * or a blank, depending on whether the indicator that caused the neighboring zero suppression was a * or some other indicator.

Some examples of pictures that contain insertion-character indicators follow:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"$9,999V.99"	1529.09	CHAR(9)	"$1,529.09"	DEC(6,2)	+1529.09
	529.09		"$0,529.09"		+0529.09
	.09		"$0,000.09"		+0000.09
	0		"$0,000.00"		+0000.00
PIC"$Z,ZZZV.ZZ"	1529.09	CHAR(9)	"1,529.09"	DEC(6,2)	+1529.09
	529.09		"$ØØ529.09"		+0529.09
	.09		"$ØØØØØ.09"		+0000.09
	0		"ØØØØØØØØØ"		+0000.00
PIC"$$,$$$V.$$"	1529.09	CHAR(9)	"$1,529.09"	DEC(6,2)	+1529.09
	529.09		"ØØ$529.09"		+0529.09
	.09		"ØØØØØ$.09"		+0000.09
	0		"ØØØØØØØØØ"		+0000.00

Each picture uses two insertion-character indicators: one comma and one period. The examples have the following features:

- Since the only digit indicators in the first picture are 9 indicators, no zero suppression occurs. Since no zero suppression occurs, the insertion characters are never suppressed.

- In the second and third pictures, zero suppression occurs. When the first digit is zero, it is suppressed and, as a result, the comma that follows is also suppressed.

- The period insertion character is never suppressed because it appears to the right of the V indicator. The V indicator stops zero suppression and therefore also stops suppression of insertion characters.

- When zero is assigned to the second or third picture, the entire character string is suppressed; this occurs because all of the digit indicators are zero-suppression indicators. However, it is not considered to be a part of the process of suppression of leading zeros.

In this example, the period is discussed as an insertion character; earlier, in the discussion of the decimal-point indicator V., the period was considered to be a part of a two-character indicator. These two views are consistent with one another, as the discussion of the "Arithmetic Decimal-Point Indicator", given later in this section, will show.

Since there is no restriction on the way insertion-character indicators are placed in a picture, a considerable variety of effects can be achieved. Examples are:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"99/99/99"	12075	CHAR(8)	"01/20/75"	DEC(6)	+012075.
PIC"99.99.99"	12075	CHAR(8)	"01.20.75"	DEC(6)	+012075.
PIC"9,999,999"	1234567	CHAR(9)	"1,234,567"	DEC(7)	+1234567.
PIC"9B999B999"	1234567	CHAR(9)	"1Ø234Ø567"	DEC(7)	+1234567.
PIC"999BB9/9"	51034	CHAR(8)	"510ØØ3/4"	DEC(5)	+51034.

The purpose of the insertion characters is to permit various special notations for values. Insertion characters should be used conservatively, and tricks should be avoided. The last example, which presumably converts an assigned integer into a three-digit integer and a fraction, is the sort of trick that should be avoided.

THE ARITHMETIC DECIMAL-POINT INDICATOR

A fixed-point picture can contain an arithmetic decimal-point indicator, which is defined as follows:

V does not have a corresponding character position and thus does not contribute a character to the character-string value. In the arithmetic interpretation, the indicator determines the position of the decimal point.

The V indicator can occur no more than once in a picture and, except for intervening insertion-character indicators, it must be adjacent to a digit indicator. Usually the V indicator is used in conjunction with the . insertion-character indicator and the pair is thought of as a single indicator as described earlier under "The Decimal-Point Indicator".

The V indicator is used separately in order to cause the automatic scaling of values that are assigned to a pictured storage unit. Examples are:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"999V99"	180.98	CHAR(5)	"18098"	DEC(5,2)	+180.98
PIC"999.99"	180.98	CHAR(6)	"001.80"	DEC(5,0)	+00180.
PIC"9.99V99"	180.98	CHAR(6)	"1.8098"	DEC(5,2)	+180.98

In each of these examples, the character-string value represents a different number than the arithmetic value.

THE SCALE-FACTOR INDICATOR

A fixed-point picture can end with a scale-factor indicator, which is defined as follows:

F(n) does not have corresponding character positions and, therefore, does not affect the related character type of a pictured storage unit. In the arithmetic interpretation, it adds n to the scale factor implied by the picture. The arithmetic value of a picture is 10**n times the apparent value of the character string value of the picture.

Some examples of the use of the scale-factor indicator in a fixed-point picture are:

Attribute	Assigned	Char Type and Value		Arith Type and Value	
PIC"99V.999F(0)"	23	CHAR(6)	"23.000"	DEC(5,3)	+23.000
PIC"99V.999F(2)"	700	CHAR(6)	"07.000"	DEC(5,1)	+0700.0
PIC"99V.999F(-1)"	.04	CHAR(6)	"00.400"	DEC(5,4)	+0.0400

The key to these examples is the fact that, for each of the three examples, the same sequence of digits appear in the character-string value and the arithmetic value. Once the related arithmetic data type has been correctly determined, it places the decimal point in the arithmetic value.

Floating-Point Pictures

A floating-point picture is composed of a sequence of four parts, as follows:

1. The mantissa, which can be any fixed-point picture that does not have a sign indicator after the digit indicators and does not have a dollar indicator.

2. A floating-point indicator, with E or K, as defined later.

3. The exponent, which is a severely restricted fixed-point picture that begins with an optional sign indicator and has from one to three Z or 9 digit indicators.

4. An optional scale-factor indicator.

The floating-point indicator distinguishes a floating-point picture; that is, any picture that contains an E or K is necessarily a floating-point picture.

Most of the interpretation of a floating-point picture depends on the interpretation of the fixed-point pictures that are used as its mantissa and exponent. However, there are a few special rules:

- Unless the arithmetic value of the assigned value is zero, the exponent is chosen so that the first digit of the mantissa is not zero.

- Before an arithmetic value is assigned to a pictured storage unit that does not have enough digits for an exact representation, the rightmost digits are dropped by rounding. This contrasts with the truncation without rounding that is used for a fixed-point picture, described earlier under "The No-Suppression Indicator".

Examples of the application of these rules are:

Attribute	Assigned	Char Type and Value	Arith Type and Value
PIC"-9V.999ES99"		CHAR(10)	FLOAT DEC(4)
	-100	"-1.000E+02"	-0001.E+2
	-.01	"-1.000E-02"	-0001.E-2
	5.0025	"+5.003E+00"	+5003.E-3

THE FLOATING-POINT INDICATORS

A floating-point picture must contain one of the following indicators:

E means that an E must appear in the corresponding position of the character-string value. In the arithmetic interpretation, it separates the mantissa and the exponent.

K does not have a corresponding character position and thus does not contribute a character to the character-string value. In the arithmetic interpretation, the indicator is equivalent to E.

The following examples show the difference between the two kinds of floating-point indicators:

Attribute	Assigned	Char Type and Value	Arith Type and Value
PIC"S9V.9999ES99"	15	CHAR(11) "+1.5000E+01"	FLOAT DEC(5) +00015.E0
PIC"S9V.9999KS99"	15	CHAR(10) "+1.5000+01"	FLOAT DEC(5) +00015.E0

THE SCALE-FACTOR INDICATOR

A floating-point picture can end with a scale-factor indicator, which is defined as follows:

F(n) does not have corresponding character positions and, therefore, does not affect the related character type of a pictured storage unit. The arithmetic value of a picture is 10**n times the apparent value of the character string value of the picture.

Some examples of the use of the scale-factor indicator in a floating-point picture are:

Attribute	Assigned	Char Type and Value	Arith Type and Value
PIC"S9V.99ES99"	15	CHAR(9) "+1.50E+01"	FLOAT DEC(3) +015.E0
PIC"S9V.99ES99F(4)"	15	CHAR(9) "+1.50E-03"	FLOAT DEC(3) +015.E0
PIC"S9V.99ES99F(-3)"	15	CHAR(9) "+1.50E+04"	FLOAT DEC(3)" +015.E0

Consider the second example. The arithmetic value 15 is assigned to the storage unit. It must be multiplied by 10**-n to get the character string representation; then when it is fetched as an arithmetic value and multiplied by 10**n its value is the same as it was originally.

Complex Pictures

Pictured storage for a complex value is declared by combining any numeric picture attribute with the COMPLEX attribute. The effect of the COMPLEX attribute is to make the character-string value twice as long as the numeric picture requires, and to thus provide for the real and imaginary parts. The resulting character string is unfortunately different from other PL/I representations of a complex value because it does not have an I at the end of the imaginary part.

An example of the declaration of a pictured complex storage unit is:

Attributes	Assigned	Char Type and Value
PIC"S9V.999ES99" COMPLEX	3-21	CHAR(20) "+3.000E+00-2.000E+00"
		Arith Type and Value
		COMPLEX FLOAT DEC(4) +0003.E0-0002.E01

Non-Numeric Pictures

The design of a non-numeric picture is much simpler than that of a numeric picture. The non-numeric pictures are present so that a programmer can describe a storage unit that is designed to hold a character string and to have only a character-string interpretation. A non-numeric picture is an alternative to the use of the CHARACTER attribute.

THE NON-NUMERIC INDICATORS

A non-numeric picture is made up of any number of non-numeric indicators in any order; however, at least one X or A indicator must appear. The indicators are defined as follows:

X means any ASCII character can appear in the corresponding character position.

A means that a letter (upper or lower case) or a blank must appear in the corresponding character position.

9 means that a digit or a blank must appear in the corresponding character position.

Observe that in a numeric picture a 9 specifies a digit only, while here, in a non-numeric picture, 9 specifies a digit or a blank.

Some examples of non-numeric pictures are:

Attribute	Assigned	Char Type and Value	Arith Type and Value
PIC"XXXXX"	"AB--3"	CHAR(5) "AB--3"	(not applicable)
PIC"AAXX9"	"AB--3"	"AB--3"	(not applicable)
	"ØØØØØ"	CHAR(5) "ØØØØØ"	
	"7B--3"	(conversion)	

The last line shows that an attempt to assign a digit to a position controlled by an A indicator causes the CONVERSION condition to occur. See the section on "Condition Handling", later in this manual.

Guidelines for Pictured Storage

Pictured storage is ideal for programming applications in which the format and layout of input/output is important and calculations are relatively simple. Business programs often fit this description. Simple programs for teaching or one-shot execution also can make good use of pictured storage.

When a pictured variable is involved in complicated and repeated arithmetic calculations, some significant extra costs can accrue. The fetch of the arithmetic value of a pictured variable can be one or two orders of magnitude more costly than the fetch of the value of a similar arithmetic variable. In such a situation, there are four options:

- Accept the extra cost and keep the pictured variable. This is the approach often adopted in business programming, where the cost of input/output overshadows any computational costs.

- Keep the pictured variable, but introduce an arithmetic variable for use in calculation. In this case, the pictured variable is used for input/output, the arithmetic variable is used in calculations, and the values are assigned back and forth between the two variables when necessary. This approach is especially useful when input/output is performed in terms of records, as described later in the section on "Record Input/Output", rather than in terms of a stream.

- Drop the pictured variable, but use the picture format item in conjunction with edit-directed input/output, as described later in the section on "Stream Input/Output". This approach is similar to the previous one, except that the pictured variable is a system temporary that is controlled by the PL/I processor rather than the program.

- Drop the use of the picture entirely. In this case, input is controlled by some other mechanism of input/output, usually within the extensive facilities for stream input/output. This approach is often used in short scientific programs.

In a large program, all four of these approaches might appear as a result of the separate consideration of each variable. In that case, it becomes an advantage to have all of these options present in a single language.

ADDRESS STORAGE

PL/I permits an address to be treated as a data value. Although these values are not subject to calculation in the usual sense, they can be stored, fetched, compared with one another and, ultimately, used as addresses. The explicit manipulation of address values is an advanced feature of PL/I, and is not used at all in most applications of PL/I.

Address values are discussed earlier in the section entitled "Values". There are six types of address values, as follows:

label	The address of a statement that can be the destination of a GOTO statement
entry	The address of a statement that is an entry to a procedure
format	The address of a FORMAT statement
pointer	The address of a storage unit, expressed as a full GCOS address
offset	The address of a storage unit, expressed relative to the beginning of an area
file	The address of a file-control block, which, in turn, gives access to a GCOS file.

For each of these types of address values, there is a corresponding type of storage unit. These storage units are described here.

A label, entry, or format address is a special kind of program address. It designates not only a certain statement of a program but also a particular activation of the block in which that statement appears. This feature is significant only when a program is recursive; it is discussed later, in the section on "Procedure Invocation".

Address Attributes

The data type of an address storage unit is specified by one of the following keywords:

```
LABEL
ENTRY
FORMAT
POINTER
OFFSET
FILE
```

Sometimes other keywords are used in close association with the data type attribute in the declaration of an address variable; however, those other keywords are not a part of the storage type. For example, consider:

```
DCL L1 LABEL LOCAL;
DCL L2 LABEL;
```

Here, the storage type of both L1 and L2 is just LABEL. The keyword LOCAL is a usage attribute, and is not part of the storage type; its effect is described in the section on "Program Flow". As a second example, consider:

```
DCL P1 ENTRY(FLOAT);
DCL P2 ENTRY(CHAR(20), DIM(3) FIXED) RETURNS(CHAR(20));
```

Here, the storage type of both P1 and P2 is ENTRY. The remainder of the declarations is, again, information about the usage of the address variables.

An Abbreviation

One of the address data attributes can be abbreviated, as follows:

Keyword	Abbreviation
POINTER	PTR

An Example of Address Storage

As an example, consider the variable START whose declaration is:

```
DCL START PTR;
```

The storage type of START is

```
POINTER
```

and its storage can be diagrammed as follows:

```
          PTR
START    /____/
```

The data frame is shown as a single box, without any division into parts. A GCOS pointer is divided into several parts, but these details of structure are irrelevant from the point of view of PL/I.

AREA STORAGE

Area storage is used in connection with the PL/I facilities for storage management. In contrast to all other PL/I variables, a variable storage unit of type AREA is not used directly to store data; instead, it is a reservoir of storage that supplies storage for the allocation of other variables. The use of area storage is an advanced and specialized feature of PL/I, and many applications do not require area storage.

When an AREA variable is allocated, the program specifies only the number of words occupied by the variable. In its initial form, the AREA variable is empty. As execution of the program proceeds, variables are allocated within the area, are used, and are eventually freed. At some time after its last use, the AREA variable itself is freed. The allocation and freeing of variables is discussed in detail later, in the section on "Storage Management".

When a variable is allocated within an AREA variable, some words of the AREA variable are occupied by the allocated variable; then, when the variable is freed, the words become available again. The PL/I processor automatically keeps data that shows which words are occupied at any time; this data is kept in the AREA variable itself, and thus uses up a portion of the storage occupied by the AREA variable. Thus an AREA variable is more than just a block of storage: it is a complete storage system that is embedded in a larger system.

An important feature of an AREA variable is that it defines its own system of addresses; that is, the address of a variable that is allocated within an area can be expressed as an offset address relative to the first word of the area. When the contents of one AREA variable is copied into another AREA variable, the GCOS addresses of the variables contained in the area change, but the offset addresses relative to the area do not change. This feature of AREA variables is important wherever address variables are used in forming linked lists of data objects.

The AREA Attribute

The data type of an area storage unit has the following form:

AREA(as)

where as is the area size. The area size must be an extent; that is, it must have the form:

exp

exp REFER ref

*

where exp is an expression and ref is a reference. The first form is used in most cases; it must yield a value that can be converted to a FIXED BINARY(18) value. The second two forms are described later, in the sections on "Storage Management" and "Procedure Invocation", respectively.

The value of the area size at the time the area is allocated determines the number of words that are allocated for the area. An AREA variable is allocated just like any other variable; details are given later, in the section on "Storage Management".

The capacity of an AREA variable is always somewhat less than the area size; this is because a portion of the storage allocated for the area variable is used as the occupation record; that is, the data that shows which words of the area are in use and which are free. For example, consider the declaration:

```
DCL A1 AREA(1000);
```

According to this declaration, the AREA variable A1 occupies exactly 1000 words of GCOS storage. Suppose that many variables with the storage type

```
CHAR(40) NONVARYING
```

must be allocated in A1. This character variable occupies 10 words of storage; however, it is not possible to allocate 100 such variables in A1 because of the storage used by the occupation record. Appendix E of the PL/I User's Guide gives information for determining the exact capacity of an AREA variable.

A Default Rule

The following default rule applies to the AREA attribute:

Omitted Item	Default
area size	1024

The use of this default is not recommended. The area size is an important parameter, and should be carefully selected and explicitly given.

An Example of Area Storage

As an example, consider the following declaration:

```
DCL M AREA(2*N);
```

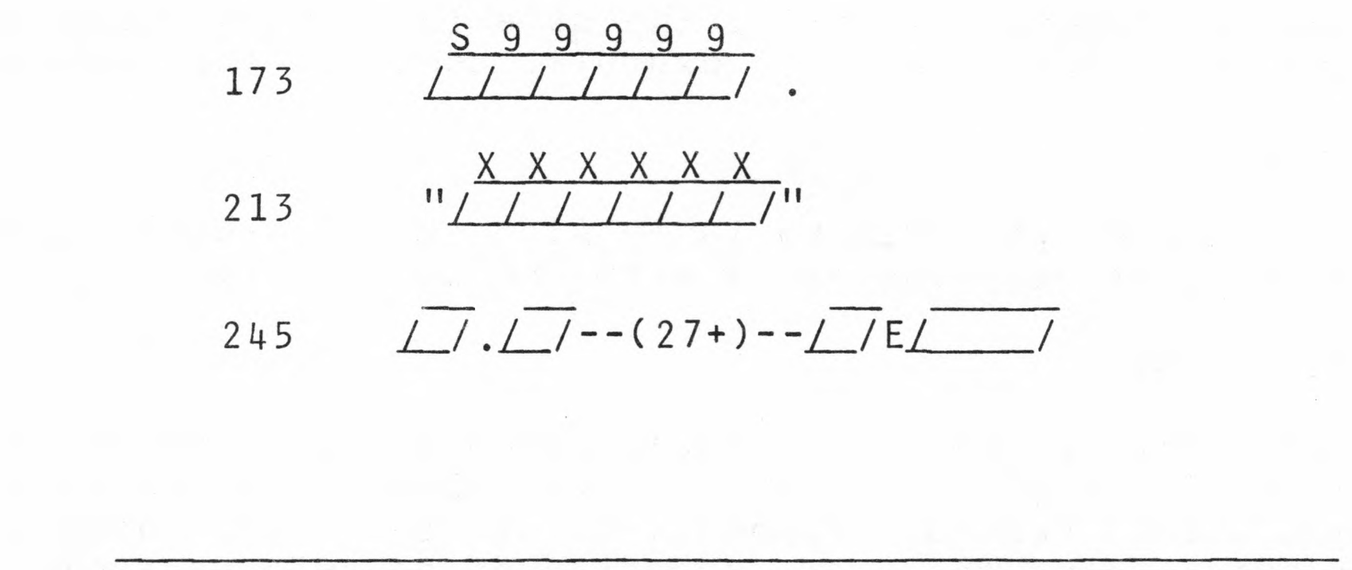
Suppose N=4096 when AREA is allocated. Then the storage type of M is:

```
AREA(8192)
```

The area variable occupies 8192 GCOS words. As program execution proceeds, variables can be allocated, used, and freed within M. Suppose that, at a particular time in program execution, three variables with the following storage types are allocated to M:

```
DEC(5)
CHAR(6)
FLOAT
```

At this time, M can be diagrammed as follows:



The number given to the left of each storage unit in the area is the word offset of the storage unit relative to the beginning of the area. The three variables shown in the diagram do not, of course, exhaust the capacity of the area.

AGGREGATE STORAGE

It is often useful to gather together a set of scalar variables, arrange them in a sequence, and treat them as a single variable. Such a variable is an aggregate. There are two kinds of aggregate, the structure and the array, and these types of variables are described here.

Once aggregates have been introduced, it is necessary to distinguish between a variable that is contained in another, larger variable and one that is not. A variable that is contained in an aggregate variable is a minor variable, and is said to be a component of the aggregate. A variable that is not contained in another variable is a major variable. Sometimes a major variable is called a level-one variable; that terminology arose from the way structures are declared. Many examples of major and minor variables are given in the discussion that follows.

Structures

A structure variable is a sequence of members. The structure itself has a name and, in addition, each of the members of the structure has its own name. When a program operates on the entire structure, the structure name alone is used as the reference. When a program operates on a member of the structure, both the structure name and the member name, separated by a period, are used as the reference. (Often the member name alone can be used, but that is just an abbreviation of the complete reference.)

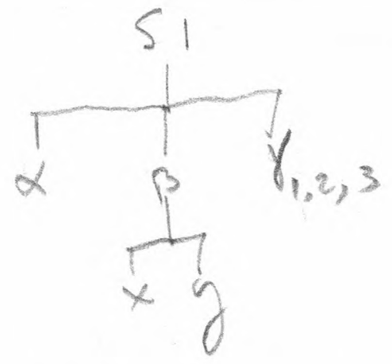
Each member of a structure can have any storage type. For example, a structure could be declared that has an arithmetic scalar as its first member, a structure of string variables as its second member, and an array of arithmetic variables as its third member.

LEVEL NUMBERS

The structure is the only part of the storage type that is not given by attributes; instead, it is given in quite a different way, by level numbers. The level number is written just before each name used in the declaration of a structure, whereas the attributes are written after the name. The level number of each member of a structure variable must be greater than the level number of the structure itself; that is how the hierarchy is indicated. It is recommended that a major variable have level number one, a member of a major variable have level number two, a member of a member should have level number three, and so on, to whatever depth is required.

For purpose of discussion, consider the following declaration of a structure:

```
DCL 01 S1,
       02 ALPHA DEC(5),
       02 BETA,
          03 X CHAR(4),
          03 Y CHAR(6),
       02 GAMMA(3) FLOAT;
```



With the help of the level numbers, 01, 02, and 03, the PL/I processor can recognize that this statement is the declaration of nine variables, as follows. The first variable is designated by:

```
S1              (designates the major structure variable)
```

The designators of the three members of S1 are:

```
S1.ALPHA        (designates a minor scalar variable)
S1.BETA         (designates a minor structure variable)
S1.GAMMA        (designates a minor array variable)
```

The designators of the two members of S1.BETA are:

```
S1.BETA.X       (designates a minor scalar variable)
S1.BETA.Y       (designates a minor scalar variable)
```

The designators of the three elements of S1.GAMMA are:

```
S1.GAMMA(1)     (designates a minor scalar variable)
S1.GAMMA(2)     (designates a minor scalar variable)
S1.GAMMA(3)     (designates a minor scalar variable)
```

Observe that this example is rather general; it shows how a member of a structure can be a scalar, a structure, or an array. (The use of an array here anticipates the description of arrays that appears later in this section; however, the array S1.GAMMA is a simple one, and its treatment should be obvious.)

Two points of programming style arise in connection with the declaration of structures.

- First, although the PL/I processor relies upon level numbers to determine the structure, a programmer relies upon the layout of the declaration. Therefore, the indentation shown in the example DECLARE statement should be used.

- Second, although the PL/I processor does not require a leading zero on a level number, the use of such a zero adds emphasis and distinguishes the level number from a computational constant.

STRUCTURE STORAGE TYPES

The storage type of a structure is obtained from the declaration in three steps:

1. Omit the name of the structure and the names of its members.

2. Normalize the level numbers.

3. Obtain the storage type of each member by appropriate methods, depending on whether the member is a scalar, a structure, or an array.

The levels of a structure are normalized by reducing them until the structure as a whole has level number one, the members of the structure each have level number two, and so on.

The members of a structure are arranged in storage in the order in which they appear in the declaration. Thus the order of the members of the structure S1 is:

```
S1.ALPHA
S1.BETA
S1.GAMMA
```

Similarly, the order of the members of S1.BETA is:

```
S1.BETA.X
S1.BETA.Y
```

As an example of the determination of the storage type of a structure, consider S1.BETA as declared in the preceding description of level numbers. The declaration was given as:

```
02 BETA,
   03 X CHAR(4),
   03 Y CHAR(6)
```

Omission of the variable names gives:

```
02,
   03 CHAR(4),
   03 CHAR(6)
```

Normalization of the level numbers gives:

```
01,
   02 CHAR(4),
   02 CHAR(6)
```

This is the desired result: the storage type for S1.BETA.

EXAMPLES

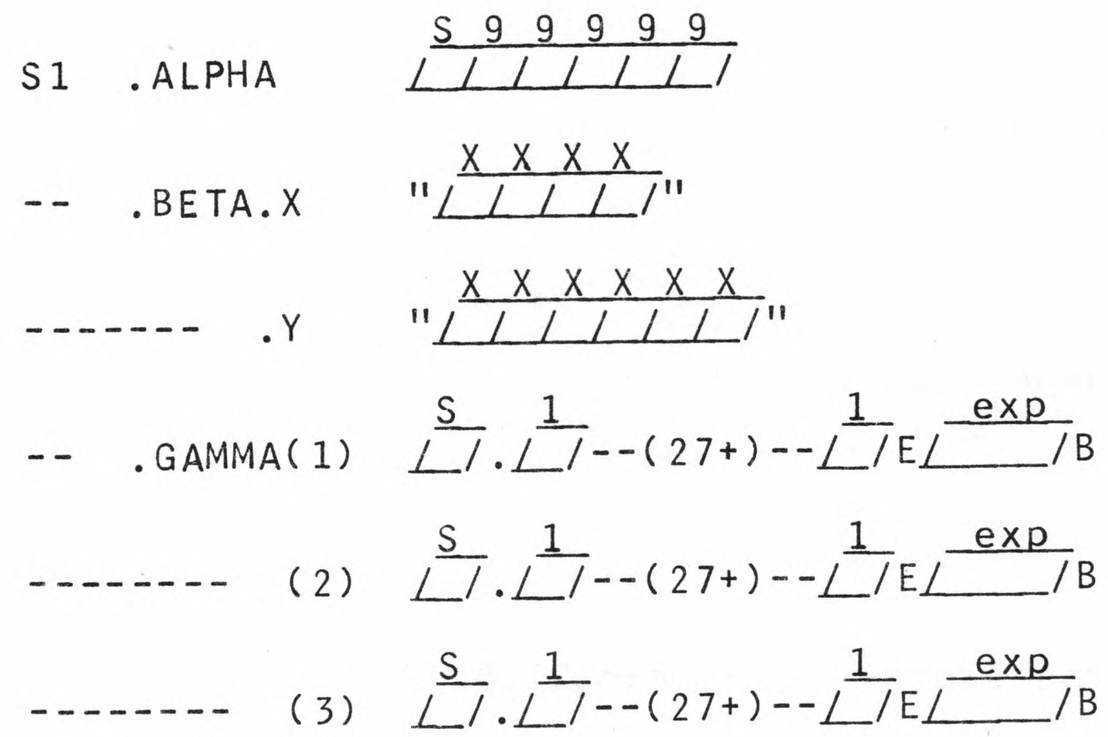
As a detailed example of storage for a structure, consider once again the variable S1, which has been used throughout this discussion of structures. Its declaration is:

```
DCL 01 S1,
       02 ALPHA DEC(5),
       02 BETA,
          03 X CHAR(4),
          03 Y CHAR(6),
       02 GAMMA(3) FLOAT;
```

This statement declares nine variables, and their storage types are as follows:

Designator	Storage Type
S1	01, 02 DEC(5), 02, 03 CHAR(4), 03 CHAR(6), 02 FLOAT
S1.ALPHA	DEC(5)
S1.BETA	01, 02 CHAR(4), 02 CHAR(6)
S1.BETA.X	CHAR(4)
S1.BETA.Y	CHAR(6)
S1.GAMMA	DIM(3) FLOAT
S1.GAMMA(1)	FLOAT
S1.GAMMA(2)	FLOAT
S1.GAMMA(3)	FLOAT

The storage for S1 can be diagrammed as follows:



Thus S1 is made up of six scalar storage units. The arrangement of these storage units on six separate lines does not imply that they occupy six words of GCOS memory. The mapping of storage units into GCOS words is done according to rules that are given later in this section, when "Alignment" is discussed. Because unused bits or bytes of storage are sometimes placed between the storage required for the storage units, the rules are not simple.

As a second example, consider the variable ACCOUNT, declared as follows:

```
DCL 01 ACCOUNT,
       02 NAME,
          03 LAST CHAR(30) VAR,
          03 MIDDLE CHAR(1),
          03 FIRST CHAR(20) VAR,
          03 TITLE CHAR(4) VAR,
       02 NUMBER PIC"9999",
       02 ADDRESS,
          03 STREET CHAR(30),
          03 LOC,
             04 CITY CHAR(20),
             04 STATE CHAR(2),
             04 ZIP PIC"99999",
       02 BALANCE PIC"$,$$$,$$9V.99",
       02 CREDIT_LIMIT PIC"$,$$$,$$9V.99";
```

This example shows how a simple financial record for a member of an organization can be handled. The following observations apply:

- The entire structure of 11 scalar components can be referred to by a single name, ACCOUNT. This reference is convenient when, for example, the structure is output as a record in a permanent file.

- The entire name of the individual who has the account can be referred to by a single designator, ACCOUNT.NAME. This reference is convenient when, for example, a list of account holders must be prepared.

- The title (Mr., Mrs., etc.) and the last name can be accessed individually by ACCOUNT.NAME.TITLE and ACCOUNT.NAME.LAST. This reference is convenient when, for example, the greeting of a letter must be prepared.

These observations are intended to emphasize the fact that the components of a structure can be handled collectively or separately, according to the requirements of each operation. A complete description of references to structures is given later, in the section on "Expressions".

Arrays

An array variable is a sequence of elements. The variable has a single name, and the individual elements are designated by giving the array name and a list of one or more subscripts. When an array variable is referenced in a program, general expressions can be given for the subscripts, and the specific subscript values are then determined each time the reference is evaluated. In this special way, data addresses can be calculated during program execution, and a single array reference can designate different elements at different times.

All elements of a given array have the same storage type. That storage type can specify a scalar or structure variable, but not an array variable. In other words, the storage type of an element of an array can specify any variable except an array. This restriction is not very serious because a problem that requires an array of arrays can always be revised to use a single, multi-dimensional array.

An array variable is declared by means of a dimension attribute. The DIMENSION attribute should be inserted just after the name of the variable. Consider the following declarations:

```
DCL M1 FLOAT;

DCL 01 M2,
       02 A FIXED,
       02 B CHAR(32);
```

These are declarations of a scalar variable and a structure variable, respectively. The first can be changed into a declaration of an array of scalars by the addition of an appropriate DIMENSION attribute, and the second can be changed into an array of structures in the same way; thus:

```
DCL M1 DIMENSION(1:20) FLOAT;

DCL 01 M2 DIMENSION(1:20),
       02 A FIXED,
       02 B CHAR(32);
```

The discussion of the DIMENSION attribute that follows gives the interpretation of these declarations and also gives the abbreviations and defaults that permit them to be written in shorter forms, namely:

```
DCL M1(20) FLOAT;

DCL 01 M2(20),
       02 A FIXED,
       02 B CHAR(32);
```

THE DIMENSION ATTRIBUTE

The DIMENSION attribute has the following form:

DIMENSION(bplist)

where bplist is the bound-pair list, and is a sequence of one or more bound-pairs separated by commas. The number of bound-pairs is the dimensionality of the attribute; it determines how many subscript positions are associated with the array. For example, consider:

```
DCL A DIMENSION(1:8,J-1:2*(N+1)) FLOAT;
```

According to this declaration, A has a dimensionality of two, and its bound-pairs are 1:8 and J-1:2*(N+1).

The purpose of a bound-pair is to specify the range for a given subscript position. Each bound-pair has one of the forms:

lb : hb

*

where lb and hb are the lower and higher bounds. The second form is described later, in the section on "Procedure Invocation". The bound-pair specifies that the subscript has the following sequence of values:

lb, lb+1, lb+2, ..., hb

Consider, again, the following example:

```
DCL A DIMENSION(1:8,J-1:2*(N+1)) FLOAT;
```

The DIMENSION attribute specifies the following range of integers for the first subscript position:

1, 2, 3, ..., 8

The second bound-pair of A depends on variables, and such variables are evaluated when the array variable is allocated or, if the variables are BASED, when the array variable is referenced. Suppose the variables are J=4 and N=2 at the time of allocation. Then the attribute specifies the following range for the second subscript position:

3, 4, 5, 6

It follows that the array named A is made up of 8*4 = 32 elements.

Each bound in a dimension attribute must be an <u>extent</u>, which has one of the following forms:

<u>exp</u>

<u>exp</u> REFER (<u>ref</u>)

where <u>exp</u> is an expression and <u>ref</u> is a reference. The first form is used in most cases; it must yield a value that can be converted to a FIXED BINARY(24) value. The second form is described later in the section on "Storage Management".

ABBREVIATIONS AND DEFAULTS

The keyword DIMENSION can be abbreviated in two ways, as follows:

<u>Keyword</u>	<u>Abbreviation</u>
DIMENSION	DIM
DIMENSION	(omit the entire keyword DIMENSION if it immediately follows the name of the variable being declared.)

Since recommended practice is to write the DIMENSION attribute just after the variable name, the keyword is usually omitted. For example, consider the declaration:

```
DCL C DIMENSION(1:12,-S:1,1:3*M-2) FIXED;
```

This declaration can be abbreviated as follows:

```
DCL C(1:12,-S:1,1:3*M-2) FIXED;
```

there are some circumstances in which the declared name is not given in the declaration, and then either the abbreviation DIM must be used or the dimension must be the first attribute. Consider the following declaration:

```
DCL P ENTRY(DIM(0:5) FIXED);
```

This declaration asserts that P is the name of a procedure that takes one argument which must be a one-dimensional array whose subscript runs from zero to five.

The lower bound can be omitted; in that case, its default value is one. That is, if hb is any valid higher bound, then the bound-pair:

```
1:hb
```

can be written as:

```
hb
```

By means of this default, the declaration of C given in the preceding paragraph can be shortened to:

```
DCL C(12,-S:1,3*M-2) FIXED;
```

ARRAY STORAGE TYPES

The bound-pairs of the DIMENSION attribute in a storage type are normalized. A normalized bound-pair is:

```
1 : hb-lb+1
```

where lb and hb are the lower and upper bounds of the original, unnormalized, bound-pair. Observe that normalization of a bound-pair causes the lower extent to be 1 while leaving the difference between the lower and upper bounds unchanged.

As examples of the normalization of DIMENSION attributes, consider the arrays D and E, declared as follows:

```
DCL D(5:N+1) FLOAT;
DCL E(2*I:2*I+3) FLOAT;
```

Suppose these arrays are allocated when N=7 and I=-1. Then the dimension attributes are:

Unnormalized	Normalized
DIM(5:8)	DIM(1:4)
DIM(-2:1)	DIM(1:4)

The unnormalized bound-pair determines the designators of the elements of each variable. The designators are:

```
D(5)     D(6)     D(7)     D(8)

E(-2)    E(-1)    E(0)     E(1)
```

The normalized bound-pair determines the storage type of each variable; therefore, both D and E have the same storage type, namely:

```
DIM(1:4) FLOAT
```

or, using the default for the lower bound:

```
DIM(4) FLOAT
```

Other examples of normalization are given in the examples of array storage that follow the next paragraph.

The elements of an array are arranged in storage in left major order. Given the designators for two elements, the order in which they appear is determined by the leftmost subscript for which the elements differ. Consider the following declaration:

```
DCL B(1:10,0:1,1:3) FLOAT;
```

The order of the elements of B in storage is:

```
B(1,0,1)
B(1,0,2)
B(1,0,3)
B(1,1,1)
B(1,1,2)
B(1,1,3)
B(2,0,1)
B(2,0,2)
B(2,0,3)
B(2,1,1)

... (and so on for 50 more elements)
```

Observe that B(1,0,1) precedes B(1,0,2) because the first two subscripts are the same and 1 comes before 2. Observe that B(1,1,2) precedes B(2,0,1) because the designators differ in their first subscript and 1 comes before 2.

EXAMPLES

As a first example of storage for an array, consider the variable A1, declared as follows:

```
DCL A1(0:2) FIXED DEC(5);
```

The storage type for A1 is:

```
DIM(3) FIXED DEC(5)
```

The storage for A1 can be diagrammed as follows:

```
                    S 9 9 9 9 9
A1  (0)             /_/_/_/_/_/_/

                    S 9 9 9 9 9
--  (1)             /_/_/_/_/_/_/

                    S 9 9 9 9 9
--  (2)             /_/_/_/_/_/_/
```

Here, as with structures, the components may be separated by unused bits or bytes of storage, and the amount of unused storage, if any, is determined by rules given later in this section, when "Alignment" is discussed.

As a second example, consider the following declaration of an array of structures:

```
DCL 01 A2(100),
       02 CODE(2) PIC"999",
       02 IDENT CHAR(5);
```

The storage type for A2 is:

```
01 DIM(100),
   02 DIM(2) PIC"999",
   02 CHAR(5);
```

The storage for A2 can be diagrammed as follows:

```
                                  9 9 9
A2  (1).CODE(1)    PIC"          /_/_/_/"

                                  9 9 9
----------  (2)    PIC"          /_/_/_/"

                                X X X X X
-----  .IDENT      "           /_/_/_/_/_/"

                                  9 9 9
--  (2).CODE(1)    PIC"          /_/_/_/"

                                  9 9 9
----------  (2)    PIC"          /_/_/_/"

                                X X X X X
-----  .IDENT      "           /_/_/_/_/_/"

... (and so on for 98 more elements)
```

As a third example, suppose an application requires a variable that is thought of as an array of arrays. Since an array of arrays is not permitted, the problem must be restated. The usual solution is to express the variable as a two-dimensional array, thus:

```
DCL A3(N,0:2) DEC(5);
```

Suppose N=2 when A3 is allocated. Then the storage type is:

```
DIM(2,3) DEC(5)
```

The storage for A3 can be diagrammed as follows:

```
A3  (1,0)   S 9 9 9 9 9
            /_/_/_/_/_/_/

--  (1,1)   S 9 9 9 9 9
            /_/_/_/_/_/_/

--  (1,2)   S 9 9 9 9 9
            /_/_/_/_/_/_/

--  (2,0)   S 9 9 9 9 9
            /_/_/_/_/_/_/

--  (2,1)   S 9 9 9 9 9
            /_/_/_/_/_/_/

--  (2,2)   S 9 9 9 9 9
            /_/_/_/_/_/_/
```

Observe that the elements of A3 are arranged in left major order.

As a fourth example consider, once again, the problem of an array of arrays. The use of a two-dimensional array is the usual solution, but there is an alternative, as given by the following declaration:

```
DCL 01 A4(N),
       02 X(0:2) DEC(5);
```

This declares an array of structures; each structure has a single member and that member is an array of scalars. Suppose, again, N=2 when A4 is allocated. Then the storage type is:

```
01 DIM(2),
   02 DIM(3) DEC(5)
```

The storage for A4 can be diagrammed as follows:

```
A4  (1).X(0)   S 9 9 9 9 9
               /_/_/_/_/_/_/

-------  (1)   S 9 9 9 9 9
               /_/_/_/_/_/_/

-------  (2)   S 9 9 9 9 9
               /_/_/_/_/_/_/

--  (2).X(0)   S 9 9 9 9 9
               /_/_/_/_/_/_/

-------  (1)   S 9 9 9 9 9
               /_/_/_/_/_/_/

-------  (2)   S 9 9 9 9 9
               /_/_/_/_/_/_/
```

Observe the similarity of the storage diagrams for A3 and A4; only the designators differ.

As a fifth and final example, consider the variable LIST, declared as follows:

```
DCL 01 LIST,
       02 COUNT FIXED,
       02 ACCOUNT(1000),
          03 NAME,
             04 LAST CHAR(30) VAR,
             04 MIDDLE CHAR(1),
             04 FIRST CHAR(20) VAR,
             04 TITLE CHAR(4) VAR,
          03 NUMBER PIC"9999",
          03 ADDRESS,
             04 STREET CHAR(30),
             04 LOC,
                05 CITY CHAR(20),
                05 STATE CHAR(2),
                05 ZIP PIC"99999",
          03 BALANCE PIC"$,$$$,$$9V.99"
          03 CREDIT_BALANCE PIC"$,$$$,$$9V.99";
```

This example picks up from the last example, ACCOUNT, that was given under "Structures". The variable LIST provides for a maximum of 1000 accounts and also provides a variable, LIST.COUNT, to record the current number of accounts. Thus this single variable can provide the complete permanent record of the financial activity of the organization.

Guidelines for Aggregates

The choice between a structure and an array is based primarily on the following considerations:

- The data type of the members of a structure can differ from one another, whereas the elements of an array must all have the same data type.
- The selection of a member of a structure must be made when the reference is written, whereas the selection of an element of an array can be performed, by the evaluation of subscript expressions, when the array reference is evaluated.

Often arrays and structures can be combined to provide a good organization for a complicated data object.

The most important efficiency consideration for aggregates arises in connection with the bounds in the declaration of an array variable. When an array is declared with bounds that are all constants, references to the array may be an order of magnitude less expensive than when bounds are variable. When the programmer has a choice, he should use constant bounds.

When an array with a variable bound appears as a member of a structure, it should appear as late in the structure as possible. The same guideline applies to a CHARACTER or BIT string with a variable maximum length and to an AREA with a variable area size. Consider the following example:

```
DCL 01 MEM,
       02 NAME(MAXCNT) CHAR(30),
       02 CNT FIXED;
```

This declaration does <u>not</u> conform to the guideline just given: the array with a variable bound comes first in the structure. A reference to NAME uses the base address of the structure; that creates no problem. However, a reference to CNT uses the base address of the structure plus the number of words occupied by NAME. Because NAME has a variable bound, the address of CNT cannot be calculated by the compiler; it must be calculated each time CNT is referenced during execution of the program. Now consider the following revision of the declaration:

```
DCL 01 MEM,
       02 CNT FIXED,
       02 NAME(MAXCNT) CHAR(30);
```

This declaration does conform to the guideline. A reference to CNT uses the base address of the structure and a reference to NAME uses the base address of the structure plus the amount of storage (one word) occupied by CNT. Thus the addresses of both members of the structure can be calculated by the compiler.

ALIGNMENT

At the beginning of this section, it was observed that every variable has a data type, an aggregate type, and an alignment type. The data type and the aggregate type determine the values that can be accommodated by a storage unit. In contrast, the alignment type affects the way a variable is laid out in hardware storage; it has no effect on the types of values that can be accommodated by the storage unit. The alignment type is given as a single attribute, either ALIGNED or UNALIGNED.

As a simple example of alignment, consider the following declaration of an array of structures:

```
DCL 01 CELL(4096),
       02 TYPE FIXED(2),
       02 CDR FIXED(14),
       02 FLAGS BIT(3),
       02 CAR FIXED(14);
```

Since no alignment attributes are given for the array, various alignments are assumed for the array, for each element of the array, and for each member of each element. The effect is that (for this particular example), each member occupies one word of GCOS storage, each element of the array occupies four words, and the entire array occupies 4*4096 words. In this form, the array occupies more storage than necessary; however, the contents of the array can be accessed efficiently.

Now consider a slightly different declaration of the same array:

```
DCL 01 CELL(4096) UNALIGNED,
       02 TYPE FIXED(2),
       02 CDR FIXED(14),
       02 FLAGS BIT(3),
       02 CAR FIXED(14);
```

This array has exactly the same capacity as before, but it is laid out in a different way. According to the default rules, the UNALIGNED attribute given for CELL applies not only to the array but also to the elements and the members of the elements. As a result (for this example), the four members of each element are packed into a single word, so the entire array occupies just 4096 words. In this form, the array occupies much less storage, but references to its components are slower.

In most cases, the alignment of variables can be ignored. PL/I has default conventions for alignment that usually provide satisfactory results. However, there are two circumstances under which alignment must be considered:

- When the amount of storage occupied by a variable must be controlled, the alignment attribute is used. A change in alignment can cause a change of more than an order of magnitude in the storage required for a variable.

- When the layout of a variable with respect to the words, bytes, and bits of storage must be controlled, the alignment attribute is used. The proper use of alignment attributes can place a member of a structure in any given field of a hardware word. Layout is important when data is prepared for communication with facilities that are outside of PL/I.

The following discussion gives the information necessary for the use of the alignment to control the amount and layout of storage for a variable. The alignment attribute is described in a way that is independent of the GCOS implementation of PL/I, and the abbreviations and defaults are given. The specific rules for the layout of a variable in GCOS storage are given in the GCOS PL/I User's Guide.

The Alignment Attribute

The alignment attribute is one of the following keywords:

```
ALIGNED
UNALIGNED
```

When a variable is ALIGNED, its storage is laid out in a way that facilitates access; that is, extra storage is used, where necessary, to permit the use of fast hardware operations to fetch or store the value of the variable. When a variable is UNALIGNED, its storage is laid out in a way that uses relatively little storage.

One way to facilitate access is to line up variables with word boundaries so that each variable occupies one or more full words; and that is the source of the keyword ALIGNED. One way to minimize the use of storage is to start each variable just where the last variable left off, so that, in some cases, a variable crosses word boundaries; and that is the source of the keyword UNALIGNED. The exact effect of the alignment attribute depends on a particular implementation of PL/I, and no more can be said of alignment in an implementation-independent way than what is said in the previous paragraph.

The alignment attribute is given in a DECLARE statement in the same way as other attributes; however, the alignment attribute for an array variable serves a double purpose. Consider the following examples:

```
DCL X(100) BIT(1) ALIGNED;
DCL 01 A(3) UNAL,
       02 M1 DEC(5) UNAL,
       02 M2 DEC(5) UNAL;
```

The alignment attribute on the first line of each of these statements applies both to an array and to each of its elements. Thus X is an ALIGNED array, each of whose elements is an ALIGNED scalar variable; and A is an UNALIGNED array, each of whose elements is an UNALIGNED structure.

Abbreviations and Defaults

The alignment attribute has just one abbreviation, but it has an elaborate default convention that permits most alignments to be handled by default. The abbreviation is:

Keyword	Abbreviation
UNALIGNED	UNAL

Now consider the default convention. When the alignment of a variable is not written explicitly in the declaration of the variable, then it is determined in two steps, as follows:

1. If the variable is contained in an aggregate variable that has an explicit alignment attribute, then the variable takes its alignment attribute from the smallest containing aggregate that has an explicit alignment attribute.

2. Otherwise, the variable takes its alignment from the following default rules:

Storage Type	Default
scalar	
arithmetic	ALIGNED
string	UNALIGNED
address	ALIGNED
area	ALIGNED
aggregate	
structure	UNALIGNED
array	(same as its elements)

The irregularity of the defaults in Step 2 reflects the intention of the designers of PL/I to provide the best choice for each case. For example, the use of ALIGNED arithmetic variables greatly increases the speed of calculations, whereas UNALIGNED string variables can save storage without much effect on the speed of access.

Some examples of this default convention follow. Consider the declaration:

```
DCL X FIXED;
```

This declaration has no explicit alignment attribute, so one must be supplied. Step 1 of the default rule does not apply because X is not contained in an aggregate. Step 2 supplies the default ALIGNED.

As a second example, consider the following declaration of a structure:

```
DCL 01 PAIR1 UNAL,
       02 I FIXED,
       02 J FIXED;
```

Step 1 of the default rule applies for both PAIR1.I and PAIR1.J. The result is as if the programmer had written:

```
DCL 01 PAIR1 UNAL,
       02 I FIXED UNAL,
       02 J FIXED UNAL;
```

As a third example, suppose that PAIR2 is declared in a similar way, but with no alignment attribute at all, as follows:

```
DCL 01 PAIR2,
       02 I FIXED,
       02 J FIXED;
```

Step 1 does not apply. According to Step 2, an equivalent declaration is:

```
DCL 01 PAIR2 UNAL,
       02 I FIXED ALIGNED,
       02 J FIXED ALIGNED;
```

Compare this example to the preceding example. Because PAIR2 is a structure, it is UNALIGNED by default; the two examples are the same in this respect. But the numbers I and J are handled differently because the alignment attribute of PAIR2 is not explicit and Step 2 applies rather than Step 1.

As a final example, consider the following declaration of a structure, which is chosen to illustrate a wide variety of defaults:

```
DCL 01 S,
       02 S1 BIT(3),
       02 S2(500) ALIGNED,
          03 A PIC"S99V.99",
          03 B CHAR(2) UNAL,
          03 C,
             04 ALPHA CHAR(2),
             04 BETA DEC(5) UNAL,
       02 S3 FIXED;
```

The application of Rule 1 supplies some of the required alignment attributes, with a result that is equivalent to the following declaration:

```
DCL 01 S,
       02 S1 BIT(3),
       02 S2(500) ALIGNED,
          03 A PIC"S99V.99" ALIGNED,
          03 B CHAR(2) UNAL,
          03 C ALIGNED,
             04 ALPHA CHAR(2) ALIGNED,
             04 BETA DEC(5) UNAL,
       02 S3 FIXED;
```

The application of Rule 2 completes the default alignments, with a result that is equivalent to:

```
DCL 01 S UNAL,
       02 S1 BIT(3) UNAL,
       02 S2(500) ALIGNED,
          03 A PIC"S99V.99" ALIGNED,
          03 B CHAR(2) UNAL,
          03 C ALIGNED,
             04 ALPHA CHAR(2) ALIGNED,
             04 BETA DEC(5) UNAL,
       02 S3 FIXED ALIGNED;
```

SECTION IV

VALUE CONVERSION

In many cases, a value cannot be placed in a storage unit as it is; instead, it must be represented in a form specified by the storage type of the storage unit. This adjustment of a value to a specified representation is the conversion of the value. The conversion operation is not necessarily simple; it may require that the given value be reinterpreted, approximated, truncated, or even rejected as unsuitable.

There are many storage types in PL/I; and if conversion from any type to any other type is allowed, then many kinds of conversion are required. The designers of PL/I did not seek to escape this consequence; they defined conversion wherever a reasonable definition could be devised. As a result, PL/I allows conversion between any types of computational values, including some conversions that are not entirely obvious, such as that of an arithmetic value into a character-string value. On the other hand, PL/I does not allow most conversions that involve non-computational values; in fact, the only conversion of this kind allowed is that between the two types of locator values, POINTER and OFFSET.

The storage type consists of an aggregate type as well as a data type; and PL/I allows some conversions of aggregate type. For example, if a scalar value is assigned to an array variable, the value is automatically converted to an array value before the assignment.

CONTEXTS THAT FORCE CONVERSION

In PL/I, if a conversion from one storage type to another is required but is not explicitly indicated, the conversion is automatically performed. This automatic conversion makes a program look simpler than it really is, and it has both advantages and disadvantages. Certainly any means to shorten a program is welcome, but unpleasant surprises can occur when a programmer misunderstands the rules for automatic conversion.

The conversion of a value is forced when it is assigned to a storage unit. The storage unit to which a value is assigned is the target, and the storage type of the target determines the kind of conversion performed. Sometimes the target is a variable name that has been explicitly declared in the program; in this case, its storage type is easily determined. In other cases, the target is a temporary storage unit provided by PL/I; and, in this case, the storage type is determined by special rules. In all cases the storage type of the target depends on the context of the computation. The ways in which context forces conversion are described at many places in this manual because they occur in many features of the language. An informal review of these contexts is given in the following paragraphs in order to show clearly the role of conversion in PL/I.

General Contexts

There are a variety of contexts in which an expression is used in a general way, and in these contexts, the storage type of the target is virtually unrestricted. These contexts are discussed in the following paragraphs.

ASSIGNMENT STATEMENTS

The most fundamental context for conversion is the assignment statement. For example, the statement:

```
X = Y;
```

can imply any of the possible conversions of PL/I, depending on the storage type of the variables X and Y. The conversion can be a trifling matter, as in the case

```
DCL X FIXED DECIMAL(8,2);
DCL Y FIXED DECIMAL(7,2);
...
X = Y;
```

Here, the only difference is that the target has one more high-order digit than the assigned value; and the conversion is always exact conversion. On the other hand, the conversion can be quite complicated, as in the case

```
DCL X(2,3) FIXED DECIMAL(8,2);
DCL Y FLOAT BINARY(25);
...
X = Y;
```

Here, the scale attribute must be converted from FLOAT to FIXED, the base attribute from BINARY to DECIMAL, the precision attribute from (25) to (8,2), and, most remarkable, the aggregate type from scalar to that for a two-dimensional array. Thus a small assignment statement can invoke a large conversion effort. A description of the assignment statement is given in the section on "Value Assignment".

ASSIGNMENT-LIKE CONSTRUCTS

In several important contexts, an assignment statement is implied in a more or less obvious way. For example, the DO group

```
DO I = 1 TO 2;
Q(I) = R(I);
END;
```

is equivalent to

```
I = 1;
Q(I) = R(I);
I = 2;
Q(I) = R(I);
```

Thus the example DO statement has within it the equivalent of an assignment statement. A full description is given in the section on "Program Flow".

There are a few other constructs that are closely related to the assignment statement. An INITIAL attribute assigns a value to a variable when the variable is allocated, as described in the section on "Storage Management". A REFER option, which is used in the declaration of some BASED variables, assigns a value to a member of a structure in which it appears, as also described in the section on "Storage Management". Finally a GET statement assigns an input value that is a character string to an input variable, as described in the section on "Stream Input/Output".

ARGUMENTS AND RESULTS

There are implied conversions in connection with the invocation of a procedure. For example, consider the following procedure:

```
F3:  PROC(A) RETURNS(FIXED BINARY(20));
        DCL (A,B) FLOAT;
        B = A**3;
        RETURN(B);
        END;
```

Suppose this procedure is invoked by the function reference

```
F3(4.58)
```

The argument of this function reference is a literal constant, 4.58, whose storage type is FIXED DECIMAL(3,2), and so it must be converted before being assigned to the parameter A whose storage type is FLOAT. Similarly, the returned value B is FLOAT, and so it must be converted before being assigned to the temporary storage for the value of the function reference whose storage type is FIXED BINARY(20). Thus conversion occurs for both argument and result in this case. A description is given in the section on "Procedure Invocation".

Just as conversion may be required for the invocation of a programmer-defined procedure, so also may conversion be required for a built-in function reference. However, the built-in functions do not restrict an argument to a single storage type but rather accept any one of a specified set of storage types. Thus the built-in functions are designed to minimize the need for conversion of values. Details are given in the sections on "Expressions" and "Operations".

There are built-in functions whose purpose is the explicit conversion of a value from one storage type to another. Consider the following program fragment:

```
DCL S CHAR(10);
DCL I FIXED;
...
S = CHAR(I,10);
```

Here the built-in function CHAR is used to convert the arithmetic value of I to a character string for assignment to S. The built-in conversion functions are not used very often, for two reasons. First, the necessary conversions are usually provided by default in PL/I. Second, the conversion functions are poorly designed and difficult to use. Full details are given under "Conversion Operations" in the section on "Operations".

Expressions are similar to built-in functions in their use of conversion. Although the operator in an expression may require conversion of its operands, the requirement is reduced in certain ways. For example, if both of the variables of the expression A+B are REAL FIXED DECIMAL, then both values are used without conversion, even though they may not have the same PRECISION attribute. Details are given in the sections on "Expressions" and "Operations".

Special Contexts

There are many contexts in which the value of an expression is required for a special purpose, and in these contexts the value is converted to a specific storage type. For example, consider the assignment statement:

```
X(2*I+K) = 0;
```

In this statement the expression '2*I+K' appears in a context that requires a subscript; accordingly, its value is converted to an integer. As a second example, consider the output statement:

```
PUT LIST(ALPHA);
```

In this statement, the expression ALPHA appears in a context that requires a value that can be placed in the output stream; accordingly, it is converted to a character-string.

The distinctive characteristic of a special context is that the storage type of the target is implicit; that is, it is implied by the use to which the given value is put rather than being given by means of a declaration. In order to provide an overall view, the special contexts are listed here; they are classified according to the storage types of the implied targets: integer, character-string, bit-string, and locator.

INTEGER TARGETS

When a special context requires that the value of a given expression be converted to an integer, the target storage type is:

REAL FIXED BINARY(p,0)

The number-of-digits, p, is always sufficiently large to accommodate any value that is valid in the given context. The integer contexts are as follows:

- The maximum-length expression in a CHARACTER or BIT attribute, as described in the section on "Value Storage"
- The area-size expression in an AREA attribute, as described in the section on "Value Storage"
- Each of the array-bound expressions in a DIMENSION attribute, as described in the section on "Value Storage"
- The expression in a POSITION attribute, as described in the section on "Storage Management"
- Each replication-factor expression in an INITIAL attribute, as described in the section on "Storage Management"
- Each subscript expression in a subscripted variable reference, as described in the section on "Expressions"
- The expression in a PAGESIZE or LINESIZE option in an OPEN statement for a STREAM file, as described in the section on "Stream Input/Output"
- The expression in a SKIP or LINE option in a GET or PUT statement, as described in the section on "Stream Input/Output"
- Each expression in a format specification list, as described in the section on "Stream Input/Output"
- The expression in an IGNORE option in a READ statement, as described in the section on "Record Input/Output"
- The arguments of some built-in functions (for example, the second and third arguments of SUBSTR), as described in the section on "Operations"

CHARACTER-STRING TARGETS

When a special context requires that the value of a given expression be converted to a character-string, the target storage type is:

CHARACTER(ml)

The maximum-length, ml, is determined from the storage type of the given expression. The character-string contexts are as follows:

- The expression in a TITLE option, as described in sections on "Stream Input/Output" and "Record Input/Output"
- Each expression that supplies an output value in a PUT statement, as described in the section on "Stream Input/Output"
- The expression in a STRING option in a GET statement, as described in the section on "Stream Input/Output"
- The expression in a KEY or KEYFROM option, as described in the section on "Record Input/Output"

BIT-STRING TARGETS

When a special context requires that the value of a given expression be converted to a bit-string, the target storage type is:

BIT(ml)

The maximum length, ml, is determined from the storage type of the given expression. There are two bit-string contexts, as follows:

- The test expression in an IF statement, as described in the section on "Program Flow"
- The test expression in a WHILE option in a DO statement as described in the section on "Program Flow"

LOCATOR TARGETS

In one special context, the value of an expression must be converted to a locator target. In this case, the target storage type is:

POINTER

and the given expression must already be POINTER or OFFSET. The locator context is:

- The locator qualifier expression in a locator qualified reference, as described in the section on "Expressions"

DATA TYPE CONVERSION

Rules for all possible conversions of the data type of a scalar are given in the following paragraphs. Arithmetic, character-string, bit-string, and locator targets are considered.

Pictured strings are not discussed, either as targets or as the source of the value to be considered. This omission reflects the fact that when a pictured-string storage unit is accessed, it is always considered to have an arithmetic or ordinary character-string value. An earlier section, "Value Storage", gives the relation between a pictured storage unit and its arithmetic or character-string value.

When a conversion is attempted that could be an error, an appropriate PL/I condition occurs. The occurrence of such conditions are mentioned in the following rules, but a discussion of their significance is deferred until later in this section, under the heading "Conditions for Conversions".

Arithmetic Targets

The rules for converting a scalar computational value to a given arithmetic data type follow. This is the most important kind of conversion because it includes the conversion from one type of arithmetic value to another. The conversion is performed in four steps, as follows:

1. Reinterpretation. An arithmetic value is obtained from the given value as follows:

   a. If the given value is arithmetic, then no reinterpretation is required.

   b. If the given value is a character-string, then it is reinterpreted as an arithmetic value. The string must have one of the following forms:

      sign arithmetic-constant
      sign arithmetic-constant sign arithmetic-constant i
      null-string

      The first form represents a real value, the second a complex value, and the third a zero real value. The initial sign can be omitted if it is plus. The arithmetic constant is defined in the section on "Expressions". Blanks can occur in the character-string only before or after the entire value representation, not within it. If the character-string does not satisfy all of these conditions, it cannot be interpreted as an arithmetic value and the CONVERSION condition occurs.

   c. If the given value is a bit-string, then it is reinterpreted as an arithmetic value by treating the sequence of bits as the binary representation of a positive integer, with the rightmost bit taking the unit's position.

2. Representation. The value is represented according to the scale and base attributes of the target data type, but with the mode and precision required by the value itself. The specific rules are as follows:

   a. The mode is COMPLEX or REAL depending on whether the value does or does not have an imaginary part. If the value has only an imaginary part, zero is assumed for the real part.

   b. The scale and base are those of the target data type.

   c. The precision (number of digits and scale factor) is whatever is necessary to represent the given value. Some values cannot be represented exactly, and for such values low-order digits that do not affect the final result (obtained after the "Approximation" step) are omitted.

3. Approximation. If necessary, the value is approximated. The operation depends on the mode, scale, and precision of the target data type, as follows:

   a. A FIXED value representation is adjusted to have the number of fractional digits that is specified by the target precision. This may require the addition of zeros at the right or the truncation of low-order digits. No rounding occurs.

   b. A FLOAT value representation is adjusted to have the number of mantissa digits that is specified by the target precision. This may require the addition of zeros or the truncation of low-order digits. Rounding occurs in the case of FLOAT values only.

   c. A COMPLEX value representation is adjusted to the target mode. If the target mode is REAL, then this requires the truncation of the imaginary part of the value representation.

4. Range Check. The magnitude of the value is checked to determine whether or not it can be accommodated by the target data type. The following cases apply:

   a. A FIXED value representation is adjusted to have the high-order digits that are specified by the target precision. This may require the addition of zeros on the left or the truncation of high-order digits. If truncation of nonzero high-order digits occurs, then the SIZE or FIXEDOVERFLOW condition occurs.

   b. A FLOAT value representation has its exponent checked. If the exponent is greater than 127, the OVERFLOW condition occurs. If the exponent is less than -128, the UNDERFLOW condition occurs.

This concludes the rules for conversion to an arithmetic target. A good way to learn such a set of rules is to extract from them those features that are not obvious. What is "not obvious" depends on the reader, but for the rules just given, the following items might be selected:

- Blanks can appear before or after but not within a character-string representation of an arithmetic value.

- A character string that is all blanks is interpreted as the arithmetic value zero.

- When an arithmetic value is approximated, rounding occurs only if the target is FLOAT.

- When a COMPLEX value is converted to REAL, the imaginary part is discarded without the occurrence of a condition to report a possible error.

EXAMPLES OF ARITHMETIC TO ARITHMETIC CONVERSION

As a first example, suppose the given value is the character string "25.97181ÞÞÞ" and the target data type is REAL FLOAT DECIMAL(5). The first step in the conversion is the reinterpretation of the string value to yield an arithmetic value, giving 25.97181. The second step is the representation of the value according to the target scale and base attributes, which are FLOAT and DECIMAL, giving +2597181E-5. The third step is the approximation of the value as specified by the target precision, which is 5, giving +25972E-3. The fourth step is the range check which, in this case, determines that the exponent is in the required range of -128 through +127. The final result of the conversion, +25972E-3 is a valid value representation for the given data type.

The following examples show the conversion of an arithmetic value for an arithmetic target:

Given Value	Target Type	Exact Representation	Result Value
17.876	FIXED DEC(5)	+17.876	+00017.
17.876	FIXED DEC(5,1)	+17.876	+0017.8
17.876	FIXED DEC(5,3)	+17.876	+17.876
17.876	FIXED DEC(5,4)	+17.876	(SIZE)
131	FIXED BIN(12)	+10000011.B	+000010000011.B
131	FIXED BIN(12,2)	+10000011.B	+0010000011.00B
131	FIXED BIN(12,5)	+10000011.B	(SIZE)
-6.9	FIXED BIN(9,6)	-110.1110011...B	-110.111001B
-6.9	FIXED BIN(9)	-110.1...B	-000000110.B
5.638	FLOAT DEC(5)	+5638.E-3	+05638.E-3
5.638	FLOAT DEC(3)	+5638.E-3	+564.E-2

The first group of examples (for the given value 17.876) show the effect of gradually increasing the scale factor of the target until, finally, there are not enough integer digits and the SIZE condition occurs. The second group is a similar sequence for a binary target. The third group shows the handling of a value, -6.9, that cannot be expressed exactly in binary representation; the "..." in the exact representation means "digits that do not affect the final result". The last group shows a DECIMAL FLOAT target; and includes the only example in which rounding occurs when low-order digits are truncated.

EXAMPLES OF STRING TO ARITHMETIC CONVERSION

The following examples illustrate the conversion of character-string or bit-string values to arithmetic targets.

Given Value	Target Type	Reinterpretation	Result Value
"-8.92ƀƀƀ"	FIXED DEC(5)	-8.92	-00008.
"-ƀ8.92ƀƀƀ"	FIXED DEC(5)	(CONVERSION)	
"ƀƀƀ"	FIXED DEC(5,2)	0	+000.00
""	FIXED DEC(5,2)	0	+000.00
23I	COMPLEX DEC FLOAT(5)	0+23I	+00000.E0+00023.E0I
23I	DEC FLOAT(5)	0+23I	+00000.E0
"8.23E-2	FIXED DEC(5,2)	.0823	+000.08
"8.23-2"	FIXED DEC(5,2)	(CONVERSION)	
"8.23E-127"	FLOAT DEC(3)	823E-129	(UNDERFLOW)
"101"B	FIXED DEC(4,1)	5	+005.0
""B	FIXED DEC(4,1)	0	+000.0
"0000000"B	FIXED DEC(4,1)	0	+000.0

The first group of examples shows ways in which blanks can and cannot be used in a character string that is to be converted into an arithmetic value. The second group shows how a complex value is filled out or truncated as the target requires. The third group shows how the wrong format for a floating-point value causes the CONVERSION condition to occur and also shows how the conversion of a number that appears to be in range can cause the UNDERFLOW condition to occur. The final group shows the conversion of bit-string values to arithmetic values.

Character-String Targets

The conversion of a scalar computational value to a character-string value is performed according to one of the following three rules:

1. Given Character-String. Suppose the given value is a character-string. Let the string type of the target be CHARACTER(n), where n is the maximum length. Then the conversion is as follows:

   a. If the length of the given value is not greater than n and the target is VARYING, then the given value (with its given length) is the result.

   b. If the length of the given value is not greater than n, as before, but the target is NONVARYING, then blanks are added at the right end of the given string until it is n characters long. The extended value is the result.

   c. If the length of the given value is greater than n, then the STRINGSIZE condition occurs.

2. Given Bit-String. Suppose the given value is a bit-string. It is reinterpreted as a character-string value by interpreting each bit as a 0 or 1 character. The resulting character-string value is then adjusted to the correct length by Rule 1, above.

3. Given Arithmetic Value. Suppose the given value is an arithmetic value. Then the following steps are performed:

   a. This step depends on scaling and base of the given value (not the target). A new number-of-digits, pc, scale-factor, qc, must be defined. If the given value is DECIMAL (p,q), then no conversion is required and:

      pc = p
      qc = q

      If the given value is FIXED BINARY (p,q), then the given value is converted to FIXED DECIMAL(pc,qc) where

      pc = MIN(CEIL(p/3.32)+1,59)
      qc = CEIL(q/3.32) (for q not negative)
      qc = -CEIL(-q/3.32) (for q negative)

      If the given value is FLOAT BINARY(p), then the given value is converted to FLOAT DECIMAL(pc) where

      pc = MIN(CEIL(p/3.32),59)

      Both of these conversions are performed according to the rules for an arithmetic target given earlier in this section. At the conclusion of this step, the given value necessarily has base DECIMAL. The mode of the value is not changed by this step.

   b. Next, the value is expressed as a character-string. Observe that the representation is different from that used for the value when it is in a storage unit.

      (1) If the value is REAL FLOAT, it is expressed as follows: a blank (for plus) or a -, followed by the first digit of the mantissa, followed by the decimal point, followed by the remaining digits of the mantissa, followed by E followed by a signed, three-digit exponent. Let pc be the number-of-digits specified by the precision of the given value. Then the mantissa has pc digits and the whole character string has length pc+7.

      (2) If the value is REAL FIXED, the rules for its representation as a character-string are complicated and are best given by examples (see below). Let pc be the number-of-digits specified by the precision of the given value. Unless the (rare) F suffix is used, the entire character-string has length pc+3.

      (3) If the value is COMPLEX, its representation is obtained by writing the two parts as REAL values according to rule (1) or (2), just given, and then concatenating them. If the imaginary part is positive, the blank that indicates its sign is changed to +. An I is added just after the imaginary part. Blanks that occur between the two parts are removed and are placed at the right end of the representation. Let lr and li be the lengths of the strings representing the real and imaginary parts. Then the entire character-string has the length lr+li+1.

   c. The character-string that results from Step b is adjusted to the correct length by Rule 1, above.

This concludes the rules for conversion to a character-string target. These rules are used chiefly in connection with data-directed and list-directed output, where they determine the format of the values in the listing. The rules are complicated, especially when the given value is binary. The following table will be useful in determining the precision of a binary value converted to a decimal value under Step 3.a:

N	CEIL(N/3.32)	N	CEIL(N/3.32)
1-3	1	37-39	12
4-6	2	40-43	13
7-9	3	44-46	14
10-13	4	47-49	15
14-16	5	50-53	16
17-19	6	54-56	17
20-23	7	57-59	18
24-26	8	60-63	19
27-29	9	64-66	20
30-33	10	67-69	21
34-36	11	70-73	22

This table includes the maximum number-of-digits for a GCOS PL/I binary value (71).

EXAMPLES OF CHARACTER-STRING TO CHARACTER-STRING CONVERSION

The following examples illustrate conversion of character-string values to character-string targets.

Given Value	Target Type	Result Value
"ABC"	CHAR(6) VAR	"ABC"
"ABC"	CHAR(6)	"ABCbbb"
"bABC"	CHAR(6)	"bABCbb"
"ABCbb"	CHAR(6)	"ABCbbb"
""	CHAR(6)	"bbbbbb"
"ABCDEFG"	CHAR(6)	(STRINGSIZE)

When the VARYING attribute is present, the length of the given value is not changed. When the VARYING attribute is not present, NONVARYING is assumed and the given value is extended to the maximum length. The examples show that blanks that are already in the given string are treated as ordinary characters. When the length of the given string exceeds the maximum length allowed by the target, the STRINGSIZE condition occurs.

EXAMPLES OF BIT-STRING TO CHARACTER-STRING CONVERSION

The following examples illustrate conversion of bit-string values to character-string targets.

Given Value	Target Type	Result Value
"0100"B	CHAR(7) VAR	"0100"
"0100"B	CHAR(7)	"0100bbb"

Observe that extension of the string occurs after its conversion to a character-string, so the added characters are blanks, not zeros.

EXAMPLES OF ARITHMETIC TO CHARACTER-STRING CONVERSION

Many examples are required to adequately illustrate the conversion of arithmetic values to character-string targets. Groups of examples will be given for each of the following types of given arithmetic values:

FLOAT DECIMAL
FIXED DECIMAL
FLOAT BINARY
FIXED BINARY
COMPLEX

The conversions can be understood in two ways. In many cases, it is enough to determine how many characters the character-string form of an arithmetic value requires. In some cases it is necessary to know the exact representation that is used. The examples illustrate both levels of knowledge.

FLOAT DECIMAL Values

The following examples illustrate conversion of FLOAT DECIMAL values to character-string targets.

Given Type	Given Value	Target Type	Result Value
FLOAT DEC(5)	-81993.E6	CHAR(14)	"-8.1993E+010ßß"
FLOAT DEC(5)	+81993.E6	CHAR(14)	"ß8.1993E+010ßß"
FLOAT DEC(5)	+81993.E6	CHAR(10)	(STRINGSIZE)

In the first two examples,

p = 5 (number-of-digits in given value)
p+7 = 12 (length of the representation)
length = 14 (length required of the target)

Observe that three digits are always provided for the exponent; this is part of a design policy that assures that the representation always takes p+7 columns, regardless of the particular given value.

FIXED DECIMAL Values

The following examples illustrate conversion of FIXED DECIMAL values to character-string targets.

Given Type	Given Value	Target Type	Result Value
FIXED DEC(4,2)	-49.62	CHAR(20) VAR	"ß-49.62"
FIXED DEC(4,2)	-00.02	CHAR(20) VAR	"ßß-0.02"
FIXED DEC(4,2)	+00.02	CHAR(20) VAR	"ßßß0.02"
FIXED DEC(4,0)	-8200.	CHAR(20) VAR	"ßß-8200"
FIXED DEC(4,0)	+0000.	CHAR(20) VAR	"ßßßßßß0"

Because the target is VARYING, the result string is not extended to 20 characters. Observe that the length of the result is always p+3 (where p is the number-of-digits of the given value); that is, although leading zeros, the plus sign, and a trailing decimal point are all suppressed, blanks are added at the left to compensate for the suppressed characters. Thus it is easy to calculate the length of the result.

When a value is FIXED DECIMAL(p,q) and the scale factor is negative (q<0) or the scale factor is greater than the number-of-digits (q>p), then the decimal point is not adjacent to a variable digit of the value; instead it is adjacent to a sequence of filler zeros, as the examples below show. Such fixed-point values are given special treatment, as follows:

Given Type	Given Value	Target Type	Result Value
FIXED DEC(4,-2)	+066700.	CHAR(20) VAR	"ØØ667F+2"
FIXED DEC(7,11)	-.ØØØØ8131885	CHAR(20) VAR	"-8131885F-11"
FIXED DEC(7,11)	-.ØØØØ0009487	CHAR(20) VAR	"ØØØ-9487F-11"

The slashed zeros are the filler zeros. In the representation of these values no attempt is made to place the decimal in the representation of the value; instead, each value is expressed as an integer with a scaling factor. The scaling factor in the representation is the negative of the scaling factor in the precision attribute.

FLOAT BINARY Values

The following examples illustrate conversion of FLOAT BINARY values to character-string targets. The target type CHAR(20) VAR is assumed in each case.

Given Type	Given Value	Intermediate Value	Result Value
FLOAT BIN(27)	+.1011001 ... E7B	+000000089.E0	"Ø8.90000000E+001"
FLOAT BIN(20)	-.1000000 ... E-9B	-9765625.E-10	"-9.765625E-004"

The '...' indicates zero digits that fill out the required precision of the binary value. The "intermediate value" is the value after it has been converted to a DECIMAL value but before it has been converted to a character-string. The data type for the intermediate value is calculated from the formulae of Rule 3.a, above. The calculation is as follows:

First Example

p = 27
p/3.32 = 8.1 ...
CEIL(p/3.32) = 9

Second Example

p = 20
p/3.32 = 6.02 ...
CEIL(p/3.32) = 7

The intermediate data types for the two examples are FLOAT DEC(9) and FLOAT DEC(7).

FIXED BINARY Values

The following examples illustrate the conversion of FIXED BINARY values to character-string targets. The target is assumed to be CHAR(20) VAR.

Given Type	Given Value	Intermediate Value	Result Value
FIXED BIN(9,0)	-001010101.B	-0085.	"ØØØØ-85"
FIXED BIN(9,0)	-111111111.B	-0511.	"ØØØ-511"

The "intermediate value" is again the value after it has been converted to a DECIMAL value and before it has been converted to a character-string. The data type for the intermediate value is calculated as follows:

Number-of-digits	Scale-factor
p = 9	q = 0
p/3.32 = 2.7 ...	q/3.32 = 0
CEIL(p/3.32) = 3	CEIL(q/3.32) = 0
CEIL(p/3.32)+1 = 4	

Thus the intermediate type is FIXED DEC(4,0). The second example shows that the formulae allowed one more digit (4 instead of 3) than were required by the largest possible value of the given data type. The example in the next paragraph shows the reason for this aspect of the design of PL/I.

The following is one more example of the conversion of a FIXED BINARY value to a character-string target.

Given Type	Given Value	Intermediate Value	Result Value
FIXED BIN(9,1)	-11111111.1B	-255.5	"Ø-255.5"

Here the data type of the intermediate value is calculated to be FIXED DEC(4,1). The full four digits of precision are required to represent the value -255.5. It thus becomes apparent that although a nine-bit integer never needs four decimal digits, a nine-bit number may need four decimal digits. In any case, the formulae given in Rule 3.a always apply.

COMPLEX Values

The following example illustrates the conversion of COMPLEX values to character-string targets.

Given Type:	COMPLEX FLOAT BIN(20)
Given Value:	-0.10000000000000000000E-1B+.11100000000000000000E1BI
Intermediate Type:	COMPLEX FLOAT DEC(7)
Intermediate Value:	-2500000E-7+1750000E-6I
Target Type:	CHAR(32)
Result Value:	"-2.500000E-001+1.750000E+000IØØØ"

Bit-String Targets

The conversion of a scalar computational value to a bit-string value is performed according to one of the following three rules:

1. Given Bit-String. The rules for adjusting the length of a bit-string to suit the target are the same as for character-string to character-string conversion with one exception: whereas a character-string is extended with blank characters, a bit-string is extended with zero-bits.

2. Given Character-String. Suppose the given value is a character-string. If every character is a 0 or a 1 character, then the character-string is reinterpreted as a bit-string by interpreting each character as a bit. The resulting bit-string is then converted to a bit-string of the required length by Rule 1, above. If the character-string contains a character other than 0 or 1, the CONVERSION condition occurs.

3. Given Arithmetic Value. Suppose the given value is arithmetic. First, the value pc is computed according to the following table:

Attributes of Given	Value of pc
FIXED BINARY(p,q)	MIN(71,MAX(p-q,0))
FIXED DECIMAL(p,q)	MIN(71,MAX(CEIL((p-q)*3.32),0))
FLOAT BINARY(p)	MIN(71,p)
FLOAT DECIMAL(p)	MIN(71,CEIL(p*3.32))

If the value of pc is zero, then the null bit-string is immediately adopted as the result of the conversion. Otherwise, the given value is converted to an intermediate value of data type REAL FIXED BINARY(pc,0). The result of this conversion is a BINARY integer value. This value is reinterpreted as a bit-string by ignoring the sign and treating each binary digit as a bit. The resulting bit-string is adjusted to the correct length by Rule 1, above.

EXAMPLES OF BIT-STRING TO BIT-STRING CONVERSION

The following examples illustrate the conversion of bit-string values to bit-string targets.

Given Value	Target Type	Result Value
"11010"B	BIT(8) VAR	"11010"B
"11010"B	BIT(8)	"11010000"B
""B	BIT(4) VAR	""B
""B	BIT(4)	"0000"B
"11010"B	BIT(4)	(STRINGSIZE)

The examples show the difference between conversion to a VARYING or a NONVARYING target. The last example shows that the STRINGSIZE condition occurs when the given string is too long for the target.

EXAMPLES OF CHARACTER-STRING TO BIT-STRING CONVERSION

The following examples illustrate the conversion of character-string values to bit-string targets.

Given Value	Target Type	Result Value
"11010"	BIT(8) VAR	"11010"B
"11010"	BIT(8)	"11010000"B
""	BIT(4) VAR	""B
""	BIT(4)	"0000"B
"11010"	BIT(4)	(STRINGSIZE)
"0120"	BIT(4)	(CONVERSION)
"101Ø"	BIT(4)	(CONVERSION)

The first five examples parallel the examples given for bit-string to bit-string conversion in the previous paragraph. The last two examples show that if a character that is not 0 or 1 appears in the given string, then the CONVERSION condition occurs.

EXAMPLES OF ARITHMETIC TO BIT-STRING CONVERSION

FIXED BINARY Values

The following examples illustrate conversion of FIXED BINARY values to bit-string targets. In each example, the target is assumed to be BIT(20) VAR.

Given Type	Given Value	pc	Intermediate Value	Result Value
FIXED BIN(5,0)	-01011.B	5	-01011.B	"01011"B
FIXED BIN(5,3)	+10.111B	2	+10.B	"10"B
FIXED BIN(5,-3)	+10111ØØØ.B	8	+10111000.B	"10111000"B
FIXED BIN(5,6)	+.Ø10111B	0	(NOT REQUIRED)	""B

The "intermediate value" is the value after it has been converted to a FIXED BINARY(pc,0) data type. In the last example, pc is zero, so the null bit-string is assumed without resort to an intermediate value. In all the examples, the result bit-string is a representation of the integer digits of the given value.

FIXED DECIMAL Values

The following examples illustrate conversion of FIXED DECIMAL values to bit-string targets. In each example, the target is assumed to be BIT(20) VAR.

Given Type	Given Value	pc	Intermediate Value	Result Value
FIXED DEC(5,2)	-034.95	10	-0000100010.B	"0000100010"B
FIXED DEC(5,6)	+.Ø81327	0	(NOT REQUIRED)	""B

The value of pc for the first example is calculated from a formula in Rule 3, above, as follows:

```
p = 5, q = 2
p-q = 3
(p-q)*3.32 = 3*3.32 = 9.96
CEIL((p-q)*3.32) = CEIL(9.96) = 10
MAX(CEIL(p-q)*3.32,0) = MAX(10,0) = 10
MIN(71,MAX(CEIL(p-q)*3.32,0)) = MIN(71,10) = 10
```

Thus the data type of the intermediate value for the first example is FIXED BIN(10,0). In the second example, pc comes out to be zero so that the null bit-string is the result without resort to an intermediate value.

FLOAT Values

The following examples illustrate conversion of FLOAT values to bit-string targets. In each example, the target is assumed to be BIT(20) VAR.

Given Type	Given Value	pc	Intermediate Value	Result Value
FLOAT BIN(10)	-.1101100100E+6B	10	-0000110110.B	"0000110110"B
FLOAT DEC(2)	+59E-1	7	+0000101.B	"0000101"B

COMPLEX Values

An example of a COMPLEX given value is not included here because the effect of the conversion of the value to an intermediate FIXED BIN(pc,0) is to discard its imaginary part and treat it as a REAL value.

Locator Targets

There are two kinds of locator values: POINTER and OFFSET. Conversion between the two kinds of locator values occurs without any restrictions or special rules. For example, in the statement

```
CUR->CELL = ALPHA;
```

the variable CUR appears as a locator qualifier and must have a locator value. If CUR is a POINTER variable, its value is used as is; but if it is an OFFSET variable, its value is automatically converted to a POINTER value.

AGGREGATE TYPE CONVERSION

All possible conversions of aggregate types are described in this section. The target aggregate type is always known when a conversion is performed. Conversion is allowed only in certain simple cases in which the aggregate type of the given value is the aggregate type of a component of the target. In some descriptions of PL/I, the conversion of an aggregate type is referred to by a special word "promotion".

The rules for converting a value to a given aggregate type follow. There are only two cases for which aggregate type conversion can occur, as follows:

1. Scalars. A scalar can be converted into any aggregate by taking the value of the scalar to be the value of each scalar component of the aggregate.

2. Structures. A structure can be converted into an array whose elements are structures of the same aggregate type as the given structure. The value of the given structure is taken to be the value of each element of the array.

When a given value differs from an aggregate target and cannot be converted by the rules just given, the program is invalid.

When the aggregate type of the given value agrees with that of the target (either from the beginning or after conversion), then each component of the given aggregate must be converted to the data type of the corresponding component of the target. The conversion of the components proceeds according to the rules already given for type conversion. Each such conversion is a distinct operation, and the conversions do not occur in any defined order.

An Example of Aggregate Conversion

The following example illustrates the complete conversion of a value of one aggregate type to a target of a different aggregate type.

```
Given Type:     01, 02 FIXED DEC(4), 02 CHAR(10) VAR
Given Value:    +0132., "HENRY"
Target Type:    01 DIMENSION(1:2), 02 FIXED BIN(10), 02 CHAR(8)

Result Value:   +0010000100.B, "HENRYbbb", +0010000100.B, "HENRYbbb"
```

There are three distinct actions associated with this conversion:

- The aggregate type is promoted from 01,02,02 (a structure with two scalar components) to 01 DIM(1:2),02,02 (an array of two structures that each have two scalar components).
- A FIXED DEC(4) value is converted to a FIXED BIN(10) target.
- A CHAR(10) VAR value is converted to a CHAR(8) target.

There is no assurance that these steps will occur in this or any other particular order; thus if a condition occurs during the conversions it is not possible to know exactly how far the conversion has progressed.

CONDITIONS FOR CONVERSIONS

The conditions that can occur during the conversion of values are described here. They are:

SIZE
FIXEDOVERFLOW
OVERFLOW
UNDERFLOW
CONVERSION
STRINGSIZE

A general discussion of conditions appears later, in the section on "Condition Handling"; only the relevance of the conditions to the conversion of values is discussed here.

A program can establish an ON unit that is executed when a particular condition occurs. If an appropriate ON unit is not established when a given condition occurs, PL/I supplies a default ON unit. For some conditions, it is not valid for an ON unit to return control to the point at which the condition occurred; and such conditions usually abort execution of the program. For other conditions, a plausible recovery action is taken when execution resumes at the point at which the condition occurred.

The SIZE and FIXEDOVERFLOW Conditions

Two conditions can occur during the conversion of a value to a fixed-point arithmetic value; they are as follows:

- The SIZE condition occurs when a value is converted to a fixed-point value and the target precision does not specify enough digits to the left of the point to accommodate the magnitude of the given value.
- The FIXEDOVERFLOW condition sometimes occurs when the SIZE condition would otherwise occur.

These conditions have the same purpose, but they are implemented in different ways. If the SIZE condition is enabled, PL/I detects every case in which the magnitude of a converted value exceeds the capacity of the target precision. For example, an attempt to assign the value 12 to a FIXED BINARY(3,0) target is detected, even though the implementation may have allowed more than three bits for the value in hardware storage. The checking has a significant cost because it often must be performed by compiled instructions rather than hardware circuitry. In contrast, the FIXEDOVERFLOW condition occurs only when the hardware detects a value which cannot fit in a hardware register.

It is invalid for an ON unit to return control to the point at which either of these conditions occurred, and so there is no simple recovery method. PL/I provides a default ON unit that writes a message on the ERROR_OUTPUT standard output file and then signals the ERROR condition. The effect of this default is to abort the execution of the program, and this is usually the appropriate response to a value that is too large.

DI OVERFLOW 6072 X27)

The OVERFLOW and UNDERFLOW Conditions

Two conditions are associated with the conversion of a value to a floating-point arithmetic value; they are as follows:

- The OVERFLOW condition occurs when the conversion produces a floating-point value whose exponent is greater than 127.
- The UNDERFLOW condition occurs when the conversion produces a floating-point value whose exponent is less than -128.

Both of these conditions usually indicate a programming error; however, the PL/I processor treats the two conditions in different ways.

The OVERFLOW condition is handled as a fatal error, similarly to the SIZE and FIXEDOVERFLOW conditions. An ON unit cannot return control to the point of occurrence, and the default ON unit writes a message and signals ERROR, thus aborting program execution. On the other hand, PL/I does not treat the UNDERFLOW condition as a fatal error. Execution can resume at the point of occurrence, and in that case PL/I sets the value in question to zero. The default ON unit writes a message, but it then returns control to the point of occurrence with a zero value.

The CONVERSION Condition

When a conversion calls upon a character-string to supply an arithmetic or bit-string value, the character-string must have contents that allow such an interpretation. When this requirement is not met, the CONVERSION condition occurs. Some examples of character-strings that cannot be converted to arithmetic targets follow, together with their corrected forms:

Incorrect	Corrected
"FIFTEEN"	"15"
"+Ø29"	"+29"
"20*3.14"	"62.8"
"-8.128422+02"	"-8.128422E+02"

The examples reflect the rules given earlier, in the definition of conversion for an arithmetic target.

An ON unit that is established for the CONVERSION condition can examine and modify the character-string that caused the CONVERSION condition; and, after taking suitable remedial action, the ON unit can return to the point of occurrence. The facilities for this action are described in the section on "Condition Handling". It is rarely possible to make a useful correction to a "bad" character-string, and the PL/I default ON unit, which writes a message on the standard output file and signals the ERROR condition, is usually the appropriate action.

The STRINGSIZE Condition

When a character-string or bit-string value is converted for a target whose length cannot accommodate the value, the STRINGSIZE condition occurs. PL/I permits recovery from this condition. If the ON unit invoked by the condition returns to the point of occurrence, exactly enough characters or bits are truncated from the right end of the string value to reduce its length to that of the target. The default ON unit does not place a message on the standard output file; it returns directly to the point of occurrence with a truncated string value.

Guidelines for Conversion Conditions

The simplest policy with respect to the conditions that occur during value conversion is to treat them all as fatal errors. PL/I partially supports this policy by providing default ON units for most of the conditions that abort program execution. The two conditions that are not handled in this way are UNDERFLOW and STRINGSIZE, and their cases must be discussed separately.

Although PL/I does not abort program execution by default when an underflow occurs, the underflow message on the standard output file should be viewed as an error report. In most computations, an underflow is just as indicative of an error as an overflow; both occur when a computation has not been properly planned. When analysis of a computation shows that an underflow could occur, the underflow should be forestalled by programmed tests that detect the development of excessively small values.

Under certain circumstances, a more advanced approach to these conditions may be required. Suppose a system for the interactive performance of calculations is to be written as a PL/I program. In such a system, the user enters commands and these commands are interpreted by the PL/I program. Suppose the user gives a command whose interpretation causes an overflow to occur. The proper response is not to abort the execution of the whole interpretive program but rather to abort the command that the user entered. For this purpose, the PL/I program could establish a special ON unit for the OVERFLOW condition.

SECTION V

PROGRAM SYNTAX

The syntax of PL/I is relatively easy to learn; it is especially uniform and reasonable in comparison to the syntax of conventional English or some earlier programming languages. This Reference Manual gives an informal definition of the syntax of PL/I, and conveys much of this definition by means of examples.

The syntax has two purposes. First, it is used to determine whether or not a given sequence of characters is syntactically valid. Some syntactically valid programs turn out to be invalid on other grounds, but the syntax does narrow the field in a reliable and effective way. Second, the syntax gives names to the components of a program. That is, the syntax defines precisely which character sequences are integers, which are expressions, and so on; therefore, these words take on an exact meaning in the discussion of a PL/I program.

This section does not give a detailed syntax for PL/I; instead, it presents the main syntactic features of PL/I. The details of syntax are given elsewhere in this manual; for example, the syntax of expressions is given in the section on "Expressions" and the syntax of the DO statement is given under "Program Flow". A detailed syntax for each PL/I statement is also given, in reference form, in the appendix, "Guide to PL/I Statements."

This section presents four views of the syntax of a PL/I program. First, it considers a program to be a sequence of characters, and describes the set of allowed characters without discussing which sequences of characters are allowed. Second, it considers a program to be a sequence of lexemes, and defines the identifiers, constants, operators, and so on. Third, it considers a program to be a sequence of statements, and describes the general syntax of statements without entering into the details of individual statement syntax. Finally, it considers the program structures of PL/I that are used to gather sequences of statements together into single program components. The section concludes with a discussion of the relation between the external procedure and a program.

CHARACTERS

A program can be viewed simply as a sequence of characters. Any of the 128 characters of the ASCII character set can appear ina PL/I program. A simple classification of those characters as they are used by PL/I follows:

letters:	A B C ... Z a b c ... z
digits:	0 1 2 ... 9
special characters:	+ - * / = > < ^ & !
	. , : ; ( ) " _ $ %
spaces:	blank tab newline newpage
strings only:	(the remaining ASCII characters)

The last item in the classification, "strings only", is included because every ASCII character, even if it has no other significance in PL/I, can appear in a character-string constant.

The set of characters required for PL/I is relatively small; and all but four or five of the characters are found on a standard typewriter. Therefore, it is convenient to type and publish PL/I programs on conventional, noncomputerized equipment.

LEXEMES

A program can be viewed as a sequence of lexemes. The division of a program into lexemes corresponds to the division of ordinary text into words, punctuation marks, and spaces. As an example of the division of PL/I text into lexemes, consider the following assignment statement:

```
ALPHA= 5.66;
```

This statement is a sequence of five lexemes, as follows: the identifier ALPHA, the operator '=', the separator (blank), the literal constant 5.66, and the punctuator ';'. A long string of terminology can be applied to a single lexeme; for example, 5.66 is, in fact, a "fixed-point decimal real arithmetic literal-constant".

In what follows, each kind of lexeme is described; there are eight kinds of lexemes, as follows:

identifier
literal constant
punctuator
operator
picture
isub
%INCLUDE
separator

After the lexemes are described, the separation rule for lexemes is given; it is the rule that governs blanks, newlines, and so on, to keep the lexemes of a program from running together. The discussion of lexemes concludes with a detailed classification of the lexemes.

Identifiers

An identifier is either a single letter or a letter followed by a sequence of characters; each character of the sequence must be a letter, a digit, a break, or a dollar sign. An identifier can be up to 256 characters long, so its length is, practically speaking, unlimited.

Often identifiers use only letters and digits. Examples of such identifiers are:

```
ALPHA   ALPHA23   ALPHA23XYZ
```

A break character is used where, in English, a hyphen would be used. The hyphen is not available in PL/I because it cannot be distinguished from a minus sign. Examples of identifiers with break characters are:

```
account_name  intake_manifold_pressure
```

Some care is necessary in choosing identifiers for external names. External names are sometimes converted by the compiler to meet requirements of the Loader, and the result could conflict with reserved system names or another external name. A detailed discussion of this question is given in Appendix F of the PL/I User's Guide, entitled "External Names".

KEYWORD VS. NAME

A particular occurrence of an identifier in a PL/I program is either a keyword or a name. When an identifier is used as a keyword, it has a specific meaning that is part of the definition of PL/I; for example, when the identifier GOTO is used as a keyword, it always specifies a transfer of control. In contrast, when an identifier is used as a name, its meaning depends on its declaration in the program; for example, the identifier X can be declared as a FIXED DEC(8) variable, a LABEL constant, or any other kind of name.

In some programming languages, the identifiers that are used as keywords are reserved for that purpose only. However, in PL/I there are so many keywords that it would be a considerable disadvantage to reserve all of them, and so PL/I does not have this restriction. Instead, the interpretation of an identifier depends on its position in the syntax of a program. Suppose, for example, that a statement begins with the identifier GOTO. The statement certainly could be a GOTO statement; for example:

```
GOTO LAB;
```

In this case, the identifier GOTO is interpreted as a keyword. However, a statement that begins with GOTO could be an assignment statement; for example:

```
GOTO = 1;
```

In this case, the identifier is a name (and it must be properly declared). Thus the identifier GOTO can be either a keyword or a name, depending on the context in which it appears.

It might be thought that there are cases in which it is difficult to determine whether an identifier is being used as a keyword or a name. This is not the case. In practice, an elementary knowledge of PL/I is sufficient to determine the interpretation of the identifiers in a program.

As an example of the interpretation of identifiers, consider the following program, in which keywords are underlined and names are not:

```
P:   PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL DATA FLOAT;
        GET DATA(DATA);
        DATA = DATA**2;
        PUT DATA(DATA);
        END;
```

(The underlining in this example is only for the purposes of discussion; in a true PL/I program, an identifier is never underlined.) In this example, P, SYSIN, and SYSPRINT are used as names, DATA is used both as a name and a keyword, and the other identifiers are used as keywords.

GUIDELINES FOR IDENTIFIERS

A programmer can use as names whatever identifiers are convenient. The choice of names does not affect the meaning of a program or its efficiency, but it does affect the readability of the program. The larger the program, the more important is the choice of name. The following suggestions apply:

- Where possible, use a long and descriptive name for a variable that is only referenced a few times. It is not much trouble to write out so few references, and the reader can understand the meaning of such a name immediately. On the other hand, use a short, abbreviated name for a variable that is referenced many times. In such a case, the resulting compactness is worth the trouble of introducing an abbreviation.

- When abbreviations are used, choose them according to some uniform rules, so that similar variables have similar names.

- Avoid using common keywords as names. When names such as

  GOTO DECLARE DCL IF THEN

  and so on are used as names, a superficial but irritating confusion is introduced. On the other hand, do use uncommon keywords as names where that is convenient. There is certainly no harm in using DFT to name a variable for the "debit final total" (or something of the sort) even though DFT is a keyword.

- Where possible, avoid using troublesome letters in identifiers. For example, the digits zero and one are troublesome because some output devices do not clearly distinguish between zero and the letter O or between one and the letter 1.

Literal Constants

There is a literal constant lexeme for each type of arithmetic and string value. The full syntax and interpretation of these lexemes are given later, in the section on "Expressions". The following is a representative set of examples of arithmetic literal constants:

Arithmetic Constant	Data Type
304	FIXED DEC(3)
3.04	FIXED DEC(3,2)
3.04E-5	FLOAT DEC(3)
3.04E-5I	COMPLEX FLOAT DEC(3)
0110001B	FIXED(7)
011.0001B	FIXED(7,4)
011.0001E-2B	FLOAT(7)
011.0001E-2BI	COMPLEX FLOAT(7)

Observe that an arithmetic constant does not begin with a sign. When a negative constant is required, it is written as two lexemes, a sign followed by an arithmetic constant.

The following is a representative set of examples of string literal constants:

String Constant	Data Type	Remark
"ABCD"	CHAR(4)	
(3)"ABCD"	CHAR(12)	means "ABCDABCDABCD"
""	CHAR(0)	means the null character string
"""HELLO,"" HE SAID."	CHAR(17)	"" counts as " in value
"11101"B	BIT(8)	
(4)"01"B	BIT(8)	means "01010101"B
""B	BIT(0)	means the null bit string

Any ASCII character can be used in a CHARACTER string constant, including such nonprinting characters as tab, newline, and so on. A string constant is a single lexeme, and is not considered to contain smaller lexemes.

Punctuators

There are six punctuator lexemes; each is given, together with its purpose, in the following table:

Punctuator	Purpose
. (period)	indicates the decimal or binary point; also, separates names in a qualified reference
, (comma)	separates items in a list of arguments, parameters, subscripts, declarations, options, and so on
: (colon)	terminates a condition prefix or a label prefix; separates the bounds of an array; and also appears in the RANGE option of a DEFAULT statement
; (semicolon)	terminates a statement
((left parenthesis)	indicates the beginning of a list, an expression, an iteration factor, and so on
) (right parenthesis)	indicates the end of a list, an expression, an iteration factor, and so on

These lexemes are used in most of the features of PL/I.

Operators

There are five kinds of operator lexemes; they are defined as follows:

Classification	Operators
arithmetic	+ - * / **
relational	= ^= < ^< > ^> <= >=
logical	^ & !
string	!!
qualifier	->

Most of the operators are defined in the section on "Operators". The only exception is the qualifier operator, which is defined in the section on "Expressions".

Pictures

The picture lexeme is a specialized lexeme that is used to specify the format of a character string. It begins and ends with a quote, but the characters that appear between quotes are restricted. An example of a picture is:

```
"$9,999.99"
```

This picture specifies a character-string of length nine that consists of a dollar sign, followed by a digit, followed by a comma, followed by three digits, followed by a period, followed by two digits. In other words, it describes a six-digit, dollars-and-cents figure.

A picture lexeme is used in only two contexts. It is used in a PICTURE attribute in the declaration of a pictured variable; this usage is described earlier, in the section on "Value Storage". Second, it is used in a picture format item in an edit-directed input/output statement; this usage is described later, in the section on "Stream Input/Output". In both cases, the picture itself is interpreted in the same way.

Isubs

The isub lexeme is a very specialized lexeme that is used only in the DEFINED attribute. It is composed of an unsigned integer followed by the three letters SUB; for example, 5SUB. Consider the following declaration of the array B:

```
DECLARE B(2,4) DEFINED A(2SUB,3*1SUB);
```

The first isub lexeme is 2SUB and means, "the value of the second subscript in a reference to B". The second isub lexeme is 1SUB and means, "the value of the first subscript in a reference to B". Thus, for example, the statement:

```
B(I-1,J) = 1;
```

is interpreted as:

```
A(J,3*(I-1)) = 1;
```

The %INCLUDE Macro

A %INCLUDE macro is a sequence of three lexemes, and has the following form:

```
%INCLUDE mn ;
```

where mn is the macro name. The macro name is an identifier of up to 32 characters in length. The %INCLUDE macro is discussed here, under "Lexemes", because it begins with a lexeme that has a special form (it begins with a percent character) and because its effect is to modify the lexical content of a program.

MACRO INTERPRETATION

A %INCLUDE macro is a command directly to the compiler; it is the only construct in the language that is not merely translated by the compiler for later execution. A %INCLUDE macro instructs the compiler to take the following steps:

1. Obtain the macro name and locate the macro that is designated by this name in the INCLUDE library file.

2. Compile the given program as if the %INCLUDE macro were replaced by a blank, followed by the contents of that macro located in Step 1, followed by a blank.

This definition of the %INCLUDE macro requires further explanation, as follows:

- When the compiler obeys a %INCLUDE macro, it does not modify any of the programmer's files. It behaves <u>as</u> <u>if</u> the given program were modified.

- The contents of an included macro can, itself, use a %INCLUDE macro, and that macro is interpreted just as if it appeared in the given program.

- In Step 2, a blank is inserted before and after the included text to keep the first and last lexeme from running into neighboring lexemes.

AN EXAMPLE MACRO

As an example of the expansion of a %INCLUDE macro, suppose that the following procedure exists:

```
P:   PROC;
        DCL ALPHA FLOAT;
        DCL BETA(20) CHAR(10);
        %INCLUDE XPOOL;
        ...
        END;
```

Each time this procedure is compiled, the compiler finds the macro XPOOL in the INCLUDE file attached by the file code .L and replaces the %INCLUDE macro by the contents of that macro. Suppose XPOOL exists and contains the following declarations:

```
DCL 01 X EXTERNAL STATIC,
       02 SIZE FIXED,
       02 TAB(1000) CHAR(10);
DCL CNT FIXED;
```

For the particular compilation under consideration, the effect is as if the procedure P contained the following material:

```
P:   PROC;
        DCL ALPHA FLOAT;
        DCL BETA(20) CHAR(10);
        DCL 01 X EXTERNAL STATIC,
               02 SIZE FIXED,
               02 TAB(1000) CHAR(10);
        DCL CNT FIXED;
        ...
        END;
```

GUIDELINES FOR MACROS

The %INCLUDE macro is often used to assist in the organization of a large program. Such a program is usually divided into several external procedures, which are written and compiled separately and then brought together for execution. The external procedures usually have certain text in common, such as the declaration of variables and routines that are used throughout the program. This common text can be handled as follows:

- Create a common library, which contains the common text. A description of the creation, maintenance and structure of an INCLUDE file is given in Section XII of the PL/I User's Guide, entitled "INCLUDE Files" and in Appendix G of the PL/I User's Guide, entitled "Structure of the INCLUDE File".

- In each external procedure, write a %INCLUDE macro that references the common library.

The important advantage of this approach is that just one copy of the common text exists, and therefore changes are made in just one place.

This use of common segments can extend through several levels of program organization. Suppose that a program is divided into subprograms and each subprogram is a set of sub-subprograms. Each sub-subprogram can use a %INCLUDE macro to reference a common segment for the subprogram. That common segment can, in turn, use a %INCLUDE macro to reference a common segment for the entire program.

The application for the %INCLUDE macro that has just been described is an important one; however, it is not the only possible use. The segment designated by a %INCLUDE macro need not be a sequence of statements; it can be any sequence of lexemes that make sense in the context in which they are used.

Separators

There are two kinds of separator lexemes: the space and the comment. They are defined as follows:

- A space lexeme is one of the following characters:

  blank
  tab
  newline

  Although these characters have different effects, each of them usually has the effect of leaving empty space in the listing of a program; that is why they are called "spaces".

- A comment lexeme has the following form:

  /*cs*/

  where cs is the comment string. The comment string is any sequence of ASCII characters that does not contain */. A comment is used to insert a message that is directed to the human reader of a program but that is ignored by the PL/I processor. It is necessary to learn whether the slash or the star comes first in a comment. One mnemonic is to suppose that the slash is used as the first (and last) character of a comment because the shape of the slash makes a comment easy to skip during program execution.

Separator lexemes are used interchangeably. That is, any sequence of one or more separator lexemes is equivalent to any other sequence of one or more separator lexemes. The role of separator lexemes in PL/I is given by the separation rules, which are described in the following paragraphs.

The Separation Rules

The subsequent sections of this manual give many informal rules for the syntax of PL/I. Each rule ultimately specifies that a given construct is a certain sequence of lexemes. However, the definitions make no mention of separators. Instead, they assume that the following separation rules are followed in all cases:

1. One or more separators can appear between any sequence of two lexemes. This rule permits the use of spaces to control layout and the addition of comments to provide explanations; thus a program can be made intelligible to a human reader.

2. One or more separators must appear between any sequence of two lexemes if the first is an identifier, literal constant, or isub and the second is also an identifier, literal constant, or isub. This rule requires the use of a separator where two lexemes might otherwise run together to form a single lexeme.

3. A separator must not appear within a lexeme. This rule prevents a lexeme from being broken up into two lexemes.

Consider an example of the application of these rules. One version of the ALLOCATE statement is defined (in the section on "Storage Management") as having the following form:

```
ALLOCATE id ;
```

where id is an identifier. This definition is satisfied by each of the following four constructs:

```
ALLOCATE BETA_5;

ALLOCATE     BETA_5     ;

ALLOCATE
    BETA_5;

ALLOCATE BETA_5 /*THE STRESS VARIABLE*/;
```

All of these statements mean exactly the same thing to the PL/I processor, but the first separation rule has been applied to produce differences that are important to a human reader.

A violation of the separation rules is quite obvious to a human reader, and that is why those rules can be given once here and then assumed to apply throughout the rest of the manual. Consider the following construct:

```
ALLOCATEBETA_5;
```

This construct is not a valid PL/I statement. It arose from the definition of the ALLOCATE statement, but the second separation rule was violated, allowing ALLOCATE and BETA_5 to run together and form a single identifier.

As a second example of the violation of the separation rules, consider the following construct:

```
ALL OCATE BETA_5;
```

This construct is not a valid PL/I statement. Once again, it arose from the definition of the ALLOCATE statement, but the third separation rule was violated, causing the keyword ALLOCATE to be broken up into two identifiers.

The Classification of Lexemes

The following list gives a complete classification for the lexemes of PL/I. It shows, for example, that there are two kinds of decimal real arithmetic literal-constant lexemes; namely, fixed-point and floating-point.

lexeme	
identifier:	X DECLARE RATE_OF_CHANGE TABLE$SUM2
literal constant	
arithmetic	
real	
decimal	
fixed-point:	59 18. 6.8923 .0030
floating-point:	8.23E+3 1E-5
binary	
fixed-point:	1101B 101.B 10011.101B .0001B
floating-point:	101.0001E+2B 1111E-5B
imaginary	
decimal	
fixed-point:	59I 18.I 6.8923I .0030I
floating-point:	8.23E+3I 1E-5I
binary	
fixed-point:	1101BI 101.BI 10011.101BI .0001BI
floating-point:	101.0001E+2BI 1111E-5BI
string	
character-string:	"ABC" "SAY ""NO""."" (5)"X" ""
bit-string:	"110"B (36)"1"B ""B
punctuators:	. , : ; ( )
operators	
arithmetic:	+ - * / **
relational:	= ^= < ^< > ^> <= >=
logical:	^ & !
string:	!!
qualifier:	->
picture:	"S999V.99" "$$$$9V.99CR"
isub:	1SUB 5SUB
%include:	%INCLUDE TABLE; %INCLUDE "RD>X>SM";
separator	
space:	(BLANK, TAB, AND NEWLINE CHARACTERS)
comment:	/* IGNORE THIS MESSAGE. */

The complete set of punctuators and operators and representative examples of other lexemes are given at the right.

STATEMENTS

A program can be viewed as a sequence of statements. The notion of the statement is one of the important features of the high-level languages. A PL/I programmer thinks of the statement as the executable unit of programming just as an assembly language programmer thinks of the hardware instruction as the executable unit of assembly language programming. Yet a single PL/I statement generally corresponds to about a half-dozen hardware instructions -- sometimes less, often more. Therefore, from a programmer's point of view, the use of PL/I instead of assembly language reduces the number of executable units in a program by a large factor.

The Statement Prefix

The following syntax rules apply to all PL/I statements:

- A statement is a prefix, followed by a statement body, followed by a semicolon.
- The prefix of a statement consists of an optional sequence of condition prefixes followed by an optional sequence of label prefixes.
- A condition prefix is a parenthesized list of condition prefix names separated by commas and followed by a colon.
- A label prefix is an identifier followed by a colon.

CONDITION PREFIXES

The purpose of a condition prefix is to alter the set of conditions that are enabled during the execution of a statement or a block. When a condition is enabled, the corresponding error checking is performed with the resulting enhancement of the debugging process. When a condition is disabled, checking for the corresponding error condition is not performed and the program executes at less cost. Consider the statement:

```
(SUBSCRIPTRANGE, NOSIZE): Z(I) = ALPHA;
```

The SUBSCRIPTRANGE condition prefix name enables the corresponding condition and, in effect, forces the processor to check to see if the value of I is outside the range of the bound of the array Z; that policy is good for debugging and bad for optimization. On the other hand, the NOSIZE condition prefix name disables the size condition and frees the processor from the responsibility for detecting a value of ALPHA whose magnitude is too large for an element of the array Z; that policy is bad for debugging but good for optimization. Further details are given in the section on "Condition Handling".

LABEL PREFIXES

The purpose of a label prefix is to declare a program address constant name and to specify its value. Three cases apply:

- If a label prefix is in a PROCEDURE or ENTRY statement, then the identifier is declared as an ENTRY CONSTANT.

- If the label prefix is in a FORMAT statement, then the identifier is declared as a FORMAT CONSTANT.

- If the label prefix is in the prefix of any statement not mentioned in the preceding cases, then the identifier is declared as a LABEL CONSTANT.

The classification just given indicates that, although the terminology "label prefix" is convenient, the identifier in a label prefix is not necessarily a label constant. Further details are given in the section on "Program Flow".

The Statement Body

There are about 25 different kinds of statements in PL/I; but the statements make extensive use of a small repertoire of components and obey some simple rules. The observations that follow are not recommended for study or memorization; instead, they are intended to assist the reader in recognizing the patterns that make the syntax of PL/I relatively easy to read.

- Keywords. Except for the assignment statement and the null statement, every statement body begins with a keyword. The complete syntax for each statement is given in Appendix A, and the entries in that appendix are arranged in alphabetical order according to the initial keyword.

- Options. The main part of most kinds of statement is a sequence of options. An option is a keyword followed by something in parentheses. An option specifies a subcommand within a statement, and can often be omitted from a statement without rendering the statement invalid (hence the name "option"). For example, consider the statement:

```
PUT FILE(BETA) SKIP(2) LIST(X,Y);
```

This statement consists of a keyword PUT and a sequence of three options. When the first two options are omitted, the statement is:

```
PUT LIST(X,Y);
```

In this statement, the missing options are treated in two different ways. In the absence of the FILE option, PL/I assumes FILE(SYSPRINT); but in the absence of the SKIP attribute, PL/I simply does not perform the SKIP action.

- Clauses. In a rather small number of cases, a clause is used instead of an option. A clause is an option without the parentheses. In a DO statement, each clause is a keyword followed by an expression. For example, consider:

```
DO I = 1 TO 20 BY 2 WHILE(U = V);
```

In this statement, TO 20 and BY 2 are clauses, but WHILE(U=V) is an option. (A common error is to write the WHILE option as a clause, without the parentheses.)

- Attributes. An attribute is a keyword that is sometimes followed by something in parentheses. It thus resembles an option; but it is used to give the declaration of an identifier rather than to give a subcommand. Attributes appear in DECLARE statements and, in a restricted way, in several other kinds of statements. For example,

```
DECLARE X REAL FIXED BINARY PRECISION(8,2);
```

  This statement contains four attributes, the last of which has a parenthesized list of two integers.

- Spaces. In a sequence of options that describe a single action (such as the opening of one file or the allocation of one storage unit), the options are separated by spaces rather than by commas. A similar rule applies to the attributes that describe a single identifier.

- Commas. Several statements use commas to express in one statement what would otherwise require several statements; for example,

```
DECLARE X DECIMAL FIXED(8), Y FLOAT;
```

  is equivalent to

```
DECLARE X DECIMAL FIXED(8);
DECLARE Y FLOAT;
```

  The punctuation used in such compound statements is always the comma (never the space), and the comma can be thought of as a "little semicolon" when it is used in this way. Although the use of one statement for several does save some writing effort, it does not change the cost of compiling or executing the statement; and there are some disadvantages to the use of a compound statement. A compound statement makes examination or editing of a program more difficult, and since compiler diagnostics are keyed to complete statements (not phrases within statements), a diagnostic for a complicated statement may be ambiguous.

The Classification of Statements

A classification of all the PL/I statements is given here. The classification is according to the principal function of each statement. There is a section in this Reference Manual for each of the eight kinds of statement given in the classification.

statement	
declaration	
DECLARE	DCL X FIXED DEC(5,2) STATIC INITIAL(1);
DEFAULT	DFT (VARIABLE & RANGE(C)) CHARACTER(1);
assignment	
(no keyword)	X(I-3) = Y*(SIN(THETA)-OMEGA);
program flow	
BEGIN	BEGIN;
END	END;
DO	DO K = 1 TO N+5 WHILE(M<0);
GOTO	GOTO LAB;
IF	IF BETA = -2 THEN Q = R-S; ELSE Q = 0;
(null statement)	;
procedure invocation	
PROCEDURE	P: PROC(X,Y) RETURNS(FLOAT);
ENTRY	Q: ENTRY(A1,A2);
END	END;
CALL	CALL PC(A-3*PHI,H3);
RETURN	RETURN(R-2*Z);
condition handling	
ON	ON ENDFILE (SYSIN) GOTO EXIT2;
REVERT	REVERT ENDPAGE (REPORT);
SIGNAL	SIGNAL ERROR;
storage management	
ALLOCATE	ALLOC ALPHA IN(TAB) SET(IP);
FREE	FREE Q;
stream input/output	
OPEN	OPEN FILE(AWC) PRINT LINESIZE(80);
CLOSE	CLOSE FILE(TEST2);
GET	GET FILE(WW) EDIT(X,Y)(P"XXX.99BBBBB");
PUT	PUT FILE(DROP) PAGE;
FORMAT	F: FORMAT(A(10),P"BBB--9V.99");
record input/output	
OPEN	OPEN FILE(A) KEYED SEQUENTIAL UPDATE;
CLOSE	CLOSE FILE(INSP);
READ	READ FILE(CUST) INTO(TAIL);
WRITE	WRITE FILE(REC) KEYFROM(X) FROM(Y);
DELETE	DELETE FILE(EMPLOYEE) KEY(SSNO);
REWRITE	REWRITE FILE(M) KEY(A3) FROM(BETA);
LOCATE	LOCATE BUF SET(P3) FILE(ARC);

A single example of each statement is given at the right. The selection of the example is necessarily arbitrary. Every example has a null prefix except for those statements that require at least one label prefix.

PROGRAM STRUCTURES

There are three PL/I constructs that group a sequence of statements into a single program structure: the group, the PROCEDURE block, and the BEGIN block. Each of these structures can be compared to the paragraph of conventional text; however, they are more powerful than the conventional paragraph. The additional power comes from the fact that any one of these program structures can, itself, be regarded as a single statement. Thus a group or block can be included in the sequence of statements encompassed in a larger group or block. This arrangement of one program structure within another is called nesting.

Groups

There is just one kind of group, the DO group. A DO group has the following form:

- A DO statement followed by
- A sequence of constructs, each of which is a statement, a group, or a block, followed by
- An END statement.

Groups and blocks must be nested; that is, any DO statement, PROCEDURE statement, BEGIN statement, ENTRY statement, or END statement that is contained in a given DO group must be part of a complete group or block that is contained in the given DO group.

The simplest application of a DO group does nothing more than gather a sequence of statements into a single executable unit. Consider, for example, the following IF statement:

```
IF X = 0
     THEN DO;
          Y = 1;
          Z = 2;
          END;
```

This statement sets the variables Y and Z only if X is zero. The DO group is treated as a single construct governed by the IF statement. If the DO and END statements are omitted, then the example becomes:

```
IF X = 0
     THEN Y = 1;
          Z = 2;
```

Now the meaning is different; the example consists of two statements, and the assignment to Z occurs regardless of whether X is zero or not. Therefore, the layout of the revised example is misleading and it should be written as follows:

```
IF X = 0
     THEN Y = 1;
Z = 2;
```

This layout makes it clear that only the assignment to Y is governed by the IF statement.

In the more complicated applications of a DO group, it not only gathers a sequence of statements into a single executable unit, but also executes the statements repeatedly. Consider, for example, the following group:

```
DO I = 1 TO 20;
     A(I) = B(N+I-1);
     END;
```

This statement performs the assignment statement 20 times. For each execution of the group, the index variable, I, is 1, 2, and so on up to 21. There are many variations in the repeating group, and these are described later, in the section on "Program Flow".

Blocks

There are two kinds of blocks: the PROCEDURE block and the BEGIN block. They have the following form:

- A PROCEDURE statement or BEGIN statement, depending on whether the block is a PROCEDURE or BEGIN block, followed by
- A sequence of constructs, each of which is a statement, a group, or a block, followed by
- An END statement.

An ENTRY statement can appear in a PROCEDURE block. Aside from this, any DO statement, PROCEDURE statement, BEGIN statement, ENTRY statement, or END statement that is contained in a given block must be part of a complete group or block that is contained in the given block; that is, groups and blocks must be nested.

The two kinds of block both define a scope for the declaration of identifiers. Within a given scope, an identifier can have a meaning that is entirely different from the meaning of the same identifier outside of the given scope. Thus a programmer can write a procedure that is intended for use within a large program without concern for conflict between his use of identifiers and that of other programmers. In addition, he can indicate by his use of scopes the distinction between a variable that is used just in a small procedure and one that is used throughout the procedure he is writing. The importance of scopes is so great that PROCEDURE blocks and BEGIN blocks are discussed together here, even though they are quite different in the way they are executed.

A PROCEDURE block is executed as a closed subroutine; that is, it is executed by means of a CALL statement or a function reference, and not when flow reaches it from the preceding statement. In contrast, a BEGIN block is executed in-line; that is, it is executed when flow reaches it from the preceding statement or by transfer of control to its BEGIN statement. (The BEGIN block in an ON statement is treated in a special way.)

Of the two kinds of block, the PROCEDURE block is by far the most important. A PL/I program is a collection of one or more procedures, and every subroutine within a program is written as a PROCEDURE block. Consider the following example:

```
P:    PROC;
         DCL (X,Y,Z) FLOAT;
         DCL (SYSIN, SYSPRINT) FILE;
         GET LIST(X,Y);
         CALL DIST(X,Y,Z);
         PUT LIST(Z);
DIST:    PROC(C1,C2,R)
              DCL (C1,C2,R) FLOAT;
              DCL SQRT BUILTIN;
              R = SQRT(C1**2+C2**2);
              END;
         END;
```

This example is a PROCEDURE block and it is a complete program. It contains a smaller PROCEDURE block that is a subroutine, and that is invoked by the CALL statement. A complete discussion of PROCEDURE blocks is given later, under "Procedure Invocation".

A BEGIN block is rarely used. It is sometimes essential in an ON statement, as described later under "Condition Handling". Usually, however, a need to gather statements can be handled better by a DO group or a PROCEDURE block. It is not unusual for a large program to be written without the use of any BEGIN blocks.

Summary of the Program Structures

The following table shows various properties of the three kinds of program structure:

	Group	BEGIN Block	Procedure
Gathers statements into a single executable unit.	yes	yes	yes
Can iterate the gathered statements.	yes	no	no
Is executed/skipped when reached by sequential flow of control.	executed	executed	skipped
Can be called as a closed subroutine.	no	no	yes
Defines a scope for the declaration of identifiers.	no	yes	yes

EXTERNAL PROCEDURES AND THE PROGRAM

Throughout most of this manual, the terms "external procedure" and "program" are used interchangeably. No harm comes from that, because a single external procedure can be executed as a program. In the following paragraphs, however, the distinction between the two terms is described in detail.

An external procedure is a procedure that is not contained in any larger PL/I construct. It is called "external" precisely because it is not contained inside any other PL/I construct. An external procedure can contain other procedures; and these latter procedures are all internal procedures.

A program is a set of one or more external procedures that are executed in concert. Each external procedure is written and compiled separately; and with the use of appropriate control cards (as described in Section V of the PL/I User's Guide, entitled "Loader"), a single executable unit will be produced.

Any program can be written as a single external procedure; and in that case, the distinction between a program and an external procedure is not very important. But there are good reasons to divide a program of medium or large size into several external procedures. The reasons are:

- In a large project involving more than one programmer, it is important to be able to compile and test parts of a program separately.
- For programs of considerable size, compilation in parts is less expensive than compilation of the whole; this is simply a fact of compiler technology. Furthermore, when a program becomes very large, it exceeds the capacity of the compiler and cannot be compiled as a whole.
- To change a single statement of an external procedure the entire external procedure must be recompiled. The smaller the external procedures, the smaller the cost of making an isolated change.

The division of a program into external procedures can be carried to an undesirable extreme. When the division results in variables being shared between the two external procedures, these variables must be EXTERNAL variables or procedure parameters; and the implementation of such variables is relatively expensive. In the absence of other guidance, external procedures should be kept between 100 and 1000 lines in length.

SECTION VI

DECLARATIONS

The use of a name in a program is a name reference and each name reference has a declaration. The declaration provides two items of information. First, it gives a set of attributes and (in the case of a structure variable) level numbers. Second, it associates the name reference with a particular block. Later sections of this manual describe how declarations are used in the interpretation of the different kinds of name references. This section describes the mechanism that supplies the declarations.

Each occurrence of a name in a program either supplies a declaration or uses a declaration. Consider the following fragment of a program:

```
DCL X FLOAT;
...
X = 5;
```

In this example, the first occurrence of X supplies a declaration, and the second occurrence of X makes use of that declaration. If the declaration of X were not supplied in this way, then the assignment statement could not be interpreted.

With one exception, the declaration of a name reference is determined by the PL/I compiler, once and for all, before a program is executed. The exception is for a variable extent: a variable array bound, a variable maximum string length, or a variable area size. To determine the declarations of the name references in a program, the compiler makes repeated use of two operations: the establishment of a declaration and the resolution of a name reference.

This section has three main parts. First, the constructs used for the establishment of declarations are described. Next, the rules for the resolution of names are presented. Finally, diagrams that show all possible declarations of names are provided and a complete classification of the attributes is given.

THE ESTABLISHMENT OF DECLARATIONS

This discussion begins with an example; then it describes the program constructs that are used in establishing declarations.

A Preliminary Example of the Establishment of Declarations

The following program contains several examples of the establishment of declarations:

```
P:    PROC;
         DCL X FLOAT;
         DCL (SYSIN,SYSPRINT) FILE;
L2:      GET LIST(X);
         CALL Q;
         IF X^ = 0 THEN GOTO L2; ELSE RETURN;
Q:       PROC;
              DCL L2 FLOAT;
              L2 = X**2 - 3*X**2;
              PUT SKIP LIST(X,L2);
              END;
         END;
```

The program has two blocks, which can conveniently be called the "outer block" and the "inner block". The declarations explicitly established in the program are as follows:

- In the outer block:

  X with the attribute FLOAT
  SYSIN with the attribute FILE
  SYSPRINT with the attribute FILE
  L2 with the attribute LABEL INTERNAL CONSTANT
  OUT with the attributes ENTRY INTERNAL CONSTANT

- In the inner block:

  L2 with the attribute FLOAT

- In an imaginary BEGIN block that encloses the entire program:

  P with the attributes ENTRY EXTERNAL CONSTANT

More is said of the "imaginary" block later in this section.

Observe that two declarations are established for the identifier L2 and that the identifier is used in two different ways. Such a double use is not attractive when the uses are so close together; but in a large program, the repeated use of a single identifier is difficult to avoid. When the subject of resolution is discussed later in this section, the way in which PL/I decides which declaration applies to a particular use of L2 is given.

Containment and Immediate Containment

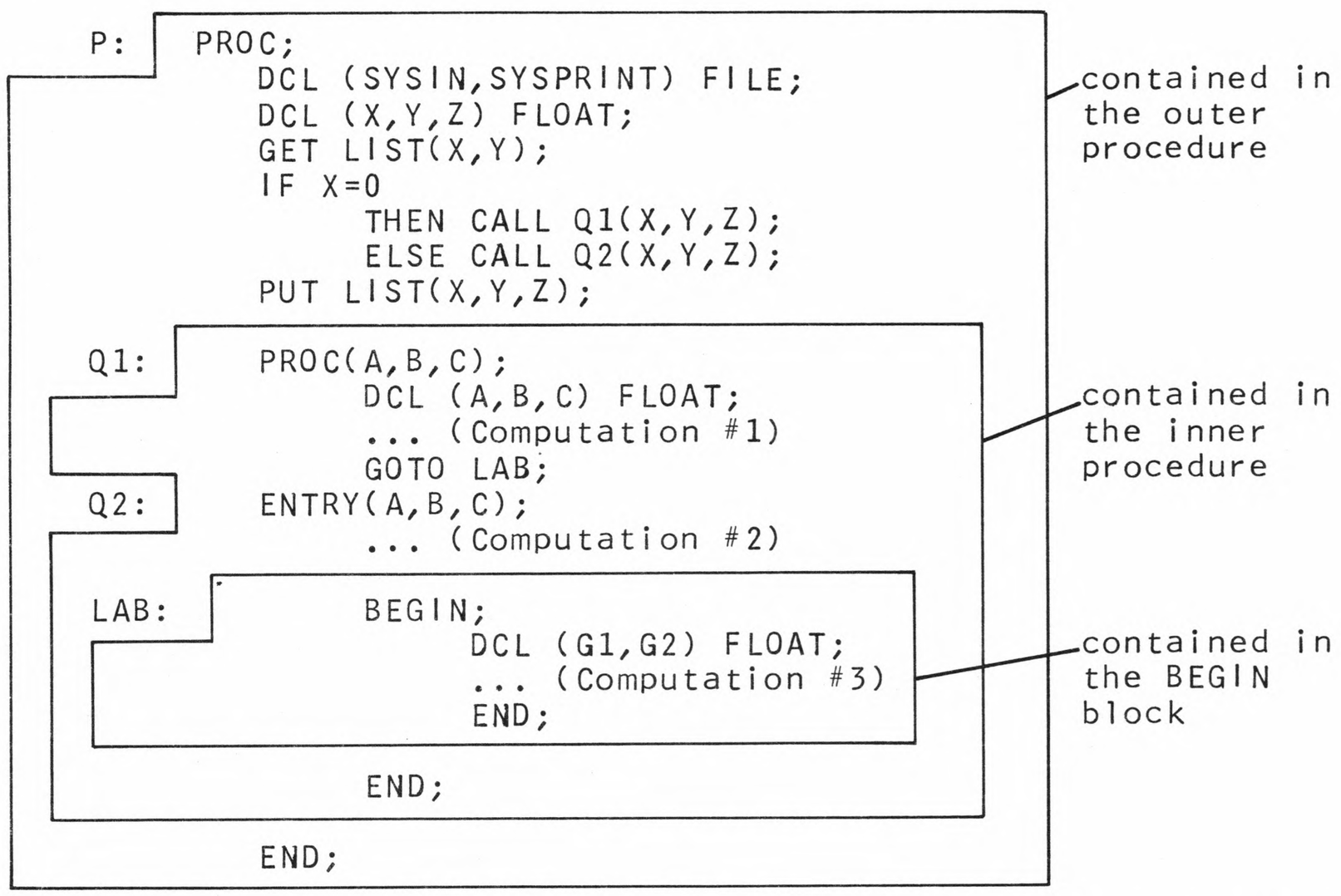
The establishment of declarations depends on the definitions of containment and immediate containment. The definitions follow:

- An occurrence of a program construct (such as a statement, a label prefix, or a name) is contained in a block if it is part of the PROCEDURE or BEGIN statement with which the block begins, part of the END statement with which the block ends, or part of any statement in between. However, there are two exceptions. First, any label prefixes in the PROCEDURE or BEGIN statement that begins a given block are not contained in the given block. Second, any label prefixes in an ENTRY statement that is part of a given block but not part of a smaller block are not contained in the given block.

- An occurrence of a program construct is immediately contained in a given block if it is contained in the given block but not in any smaller block.

In the establishment of declarations, these definitions are used to determine which block immediately contains a given DECLARE statement or label prefix.

As an example of the application of these definitions, consider the following program:



This program is composed of three blocks; they can be referred to as the "outer procedure", the "inner procedure", and the "BEGIN block". The three outlines show which portions of the program are contained in each of the three blocks.

The following observations are typical of those necessary for the establishment of declarations:

- The label prefix 'P:' is not contained in any of the three blocks, but it is contained in an imaginary BEGIN block that is considered to enclose the entire program.

- The first two DECLARE statements are immediately contained in the outer procedure.

- The label prefixes 'Q1:' and 'Q2:' are immediately contained in the outer procedure (and are not contained in the inner procedure); this is true even though 'Q2:' occurs in the midst of the inner procedure.

- The third DECLARE statement and the label prefix 'LAB:' are contained in both the outer and inner procedures, but they are immediately contained in the inner procedure only.

- The fourth DECLARE statement is contained in all three blocks; but it is immediately contained in the BEGIN block only.

The imaginary BEGIN block mentioned in connection with the declaration of P is a definitional artifice that makes the rules for the establishment of declarations simpler. The external procedures of a program are thought of as being enclosed in an imaginary BEGIN block. All of the entry constant names for the external procedures are declared in this BEGIN block.

The DECLARE Statement

The principal means for establishing a declaration is the DECLARE statement. As the keyword DECLARE suggests, the statement is devoted to supplying information. When a DECLARE statement is encountered in the course of program execution, it produces no action.

A DECLARE statement gives one or more declarations. Each declaration associates a set of attributes with a name. The declarations are established in the block that immediately contains the DECLARE statement.

The DECLARE statement is available in several forms. In many cases, a DECLARE statement is a simple declaration of a scalar or array variable name. When a structure variable is declared, a special form of the DECLARE statement is required. In order to reduce the amount of writing required, declarations can be combined and factored. These forms of the DECLARE statement are discussed in the following paragraphs.

SIMPLE DECLARATIONS

The simplest form of a DECLARE statement is the keyword DECLARE, a name, a sequence of attributes, and a semicolon. Two examples are:

```
DECLARE ALPHA_P5 DECIMAL FLOAT STATIC;
DECLARE X;
```

The effect of the first statement is to establish in the immediately containing block a declaration of ALPHA_P5 with attributes DECIMAL FLOAT STATIC. The second statement establishes in the immediately containing block a declaration of X with no attributes. In both cases, the attributes not given in the DECLARE statement are filled in by PL/I according to default rules.

The following statement declares an array variable name:

```
DECLARE BETA DIMENSION(3*N+1) FLOAT CONTROLLED;
```

When the DIMENSION attribute immediately follows the declared identifier, the keyword DIMENSION can be omitted. Thus this statement is usually written as

```
DECLARE BETA(3*N+1) FLOAT CONTROLLED;
```

The effect of this statement is to establish in the immediately containing block a declaration of BETA with the attributes DIMENSION(3*N+1) FLOAT CONTROLLED. The expression '3*N+1' is not evaluated when the declaration is established (before program execution); instead it is evaluated when storage for the array is allocated (during program execution).

STRUCTURE DECLARATIONS

Most names can be declared by means of the simple form of declaration just described. The only exception is the declaration of a structure variable name. The declaration of a structure requires the declaration of several names: one for the entire structure, others for its members, yet others for the members of each of its members, and so on. Each name is declared in a declaration clause, which consists of a level number, the name itself, and a sequence of attributes. All of the names for a given structure must be in the same DECLARE statement, and the declaration clauses are separated by commas.

Consider, for example, the following declaration of the structure variable SUBSCRIBER:

```
DECLARE 01 SUBSCRIBER EXTERNAL,
          02 NAME,
             03 FIRST CHAR(15) VAR,
             03 INITIAL_OF_MIDDLE CHAR(1),
             03 LAST CHAR(25) VAR,
          02 SERIAL_NUMBER DECIMAL(9);
```

This statement contains six declaration clauses. It establishes the following declarations in the immediately containing block:

- SUBSCRIBER, a structure with the attributes EXTERNAL and with members NAME and SERIAL_NUMBER
- NAME, a structure with no attributes and with members FIRST, INITIAL_OF_MIDDLE, and LAST
- FIRST, a scalar with attributes CHAR(15) VAR
- INITIAL_OF_MIDDLE, a scalar with attribute CHAR(1)
- LAST, a scalar with attributes CHAR(25) VAR
- SERIAL_NUMBER, a scalar with attributes DECIMAL(9)

SHORT FORMS OF DECLARATIONS

A major portion of a PL/I program is devoted to DECLARE statements, so PL/I has several features designed to shorten declare statements. The sole purpose of these features is to reduce the size of a program and thus make it easier to write and to read. The features do not have any effect on the cost of executing the program.

Abbreviations and Defaults

A useful feature is the abbreviation of the keyword DECLARE to DCL. A second feature is the provision of many abbreviations and defaults for attributes. The designers of PL/I selected the defaults to cover the most commonly required attributes. It follows that when an identifier is used in some ordinary way not many attributes need be written explicitly. For example, in the DECLARE statement:

```
DCL Q72 FLOAT STATIC;
```

the programmer has omitted the defaults REAL BINARY PRECISION(27) ALIGNED INTERNAL VARIABLE because these are supplied by default. The defaults for the storage type attributes are given in the section on "Value Storage". Other default rules are given in the section on "Storage Management".

Combining Declarations

A sequence of DECLARE statements can be combined into a single DECLARE statement. Consider the statements:

```
DCL X FLOAT;
DCL ALPHA FIXED DEC(10);
DCL Q3 FLOAT;
```

A single, equivalent statement is:

```
DCL X FLOAT, ALPHA FIXED DEC(10), Q3 FLOAT;
```

Factoring Declarations

Common attributes can be <u>factored</u> from individual declarations in a combined DECLARE statement. Another equivalent form for the DECLARE statement above is:

```
DCL (X,Q3) FLOAT, ALPHA FIXED DEC(10);
```

This feature is called factoring because it is similar to the mathematical operation of factoring out a common multiplier from a sum.

Factoring can be applied more generally than the preceding example indicates. Factoring can be applied repeatedly to the same statement. For example, the statement:

```
DCL X FLOAT STATIC, Y FLOAT STATIC, Z FLOAT;
```

can be written equivalently as:

```
DCL (X STATIC, Y STATIC, Z) FLOAT;
```

and then as:

```
DCL ((X, Y) STATIC, Z) FLOAT;
```

The result has two attributes in it instead of the original five.

Factoring can also be applied to the level numbers used in the declaration of a structure name; the only difference is that level numbers are factored to the left while attributes are factored, as before, to the right. For example, the statement:

```
DCL 01 POSITION,
       02 X FLOAT,
       02 Y FLOAT,
       02 Z FLOAT;
```

can be written as:

```
DCL 01 POSITION,
       02 (X, Y, Z) FLOAT;
```

GUIDELINES FOR DECLARE STATEMENTS

A DECLARE statement can be placed anywhere in the block in which its declaration is to be established, provided it is immediately contained in that block. It is suggested, however, that all DECLARE statements be placed immediately after the PROCEDURE or BEGIN statement that begins the block. If this convention is followed, a human reader always knows where to look for the DECLARE statements and the DECLARE statements do not clutter up the portion of the block which is devoted to executable statements.

The extensive use of the combination and factoring of DECLARE statements can make a program difficult to read, debug, and correct. Furthermore, the GCOS PL/I compiler keys its diagnostics to statements, not to individual declarations; therefore, a diagnostic message about a DECLARE statement with several declarations can be ambiguous. Some programmers avoid combination and factoring of declarations entirely and use one DECLARE statement for each variable they declare.

The use of the outline layout of the declaration of structures, as shown in the examples in this section, is recommended. The leading zero in each level number is not required in PL/I; it is a stylistic device carried over from COBOL by some PL/I programmers. It has the advantage that it distinguishes the level numbers, which do not partake in the computational activity of PL/I, from the arithmetic constants.

Label Prefixes

A second means for establishing a declaration is the label prefix. A label prefix is an identifier followed by a colon and it always occurs immediately before a statement body or another label prefix. It is useful to have a common term, "label prefix" for all uses of this construct; but, in fact, the identifier in a label prefix can be a label constant name, an entry constant name or a format constant name.

LABEL CONSTANT NAMES

When a label prefix occurs before any statement except a PROCEDURE, ENTRY, or FORMAT statement, the identifier in the prefix is a label constant name. An effect of the label prefix is to establish in the immediately containing block a declaration of the identifier as LABEL INTERNAL CONSTANT. Another, more fundamental effect, is to label the statement so that it can be the destination of a transfer of control; that effect is described later, in the section on "Flow of Control".

A parenthesized, optionally-signed integer can be used in a label prefix as a subscript to a label constant name. A given identifier can appear in several label prefixes in a single block provided each appearance has a different subscript. The effect of the set of label prefixes with the given identifier is to establish a single declaration of the identifier as LABEL INTERNAL CONSTANT DIMENSION(i1:i2), where i1 is the smallest integer used as a subscript and i2 is the largest.

As an example of the use of label prefixes to declare label constant names, consider the following procedure:

```
MES:  PROC(I);
         DCL SYSPRINT FILE;
         DCL I FIXED;
         GOTO LAB(I);
LAB(14): PUT LIST("X IS TOO BIG"); GOTO EXIT;
LAB(-3): PUT LIST("Z3 IS NEGATIVE"); GOTO EXIT;
LAB(1):  PUT LIST("A EXCEEDS X"); GOTO EXIT;
EXIT:    END;
```

The label declarations established in this procedure are:

- LAB, with attributes LABEL INTERNAL CONSTANT DIMENSION(-3,14)
- EXIT, with attributes LABEL INTERNAL CONSTANT

ENTRY CONSTANT NAMES

When a label prefix occurs before a PROCEDURE statement or an ENTRY statement, the identifier in the prefix is an entry constant name. In the event that the procedure has neither arguments nor a result, the effect of the label prefix is to establish in the immediately containing block a declaration of an identifier with the attributes ENTRY INTERNAL CONSTANT or ENTRY EXTERNAL CONSTANT. When the procedure has arguments or a result, the attributes of the arguments or the result is included in the declaration of the entry constant name.

As an example of the use of label prefixes to declare entry constant names, consider the following procedure, which is assumed to be contained in some larger block:

```
REP:  PROC(S,CNT) RETURNS(CHAR(1000) VAR);
        DCL S CHAR(1000) VAR;
        DCL CNT FIXED;
        DCL I FIXED;
        I = CNT;
        GOTO L1;
REP2: ENTRY(S) RETURNS(CHAR(1000) VAR);
        I = 1;
L1:     ...
        END;
```

The first two label prefixes establish the following declarations in the block (not shown) that immediately contains this procedure:

- REP with attributes ENTRY(CHAR(1000) VAR, FIXED) RETURNS(CHAR(1000) VAR) INTERNAL CONSTANT
- REP2 with attributes ENTRY(CHAR(1000) VAR) RETURNS(CHAR(1000) VAR) INTERNAL CONSTANT

The attributes for REP were obtained by making two changes in the PROCEDURE statement and adding the attributes INTERNAL CONSTANT. The changes are the replacement of PROCEDURE by ENTRY and the replacement of each parameter by the declaration of the parameter. The attributes for REP2 were obtained by a similar modification of the ENTRY statement. The declarations are established in the containing block because a procedure block does not contain the label prefixes of PROCEDURE and ENTRY statements in the procedure block.

The example just given shows that the declaration of an entry constant name can be long. Its purpose is to supply information needed to convert values supplied as arguments and to accept the value produced as a result. More is said of this later, in the section on "Procedure Invocation".

Suppose the procedure just given is not contained in some larger block (as was previously assumed) but is itself an external procedure. This change has two effects. First, the declarations established for both REP and REP2 are changed by replacing INTERNAL by EXTERNAL. Second, the declarations of REP and REP2 are established in the imaginary BEGIN block that encloses the program.

FORMAT CONSTANT NAMES

When a label prefix occurs before a FORMAT statement, the identifier in the prefix is a format constant name. An effect of the label prefix is to establish in the immediately containing block a declaration of an identifier with the attributes FORMAT INTERNAL CONSTANT. The role of FORMAT statements in PL/I is rather limited and is described under "Stream Input/Output".

Contextual and Implicit Declarations

A declaration that is established by a DECLARE statement or a label prefix is called an explicit declaration. PL/I allows declarations to be established in other ways, and these declarations are contextual or implicit. A contextual declaration is one that is indicated by the way in which an identifier is used; an implicit declaration is a last resort used when no other basis for establishing a declaration can be found.

AN EXAMPLE OF CONTEXTUAL AND IMPLICIT DECLARATIONS

An example of a program that uses contextual and implicit declarations is:

```
A:   PROC;
        DO I = 0 TO 90;
           PUT FILE(SYSPRINT) SKIP LIST(SIND(I));
           END;
        END;
```

This program fails to give explicit declarations for SYSPRINT, SIND, and I. The PL/I compiler assumes that declarations for these names should be established in the outermost block of the external procedure (the only block in this case) and supplies attributes as follows:

- SYSPRINT occurs in a FILE option; since only a FILE value can occur in this context, the compiler supplies the attributes FILE CONSTANT as a contextual declaration.
- SIND occurs as a function or array name; since SIND is the name of a built-in function, the compiler supplies the attribute BUILTIN as a contextual declaration.
- I occurs in a context which does not unambiguously indicate its declaration; therefore the compiler supplies no attributes, and the declaration is implicit. According to the default rules, a name with no attributes is assumed to be REAL FIXED BINARY (17,0).

It follows that the example program is equivalent to the following program, in which all identifiers are explicitly declared:

```
A:   PROC;
        DCL SYSPRINT FILE;
        DCL SIND BUILTIN;
        DCL I;
        DO I = 0 TO 90;
            PUT FILE(SYSPRINT) SKIP LIST(SIND(I));
            END;
        END;
```

The example just given shows that contextual and implicit declarations make a short PL/I program much shorter. However, in a larger, more realistic program, the use of contextual or implicit declarations can mask errors in the program. Contextual and implicit declarations are implemented in GCOS PL/I and mentioned here because they are part of Standard PL/I. However, the GCOS PL/I compiler prints a warning message for each such declaration, and it is recommended that every name in a GCOS PL/I program be explicitly declared by a DECLARE statement or a label prefix.

Special Facilities for Declaration

PL/I has two facilities that are designed to assist programmers in the declaration of names: the LIKE attribute and the DEFAULT statement. The LIKE attribute is recommended for the rather special situations to which it applies, and it is fully described here. In contrast, the DEFAULT statement is not recommended, and only a few examples are given here rather than a full definition.

THE LIKE ATTRIBUTE

The LIKE attribute asserts that the members of a given structure have the same declarations as the members of some other structure. For example, consider the LIKE attribute in the following program:

```
P:   PROC;
        DCL 01 ALPHA(1000) EXTERNAL STATIC,
               02 X FLOAT,
               02 Y CHAR(16);
        DCL 01 BETA LIKE ALPHA;
        ...
        END;
```

The second DECLARE statement is equivalent to:

```
        DCL 01 BETA,
               02 X FLOAT,
               02 Y CHAR(16);
```

The Form of the LIKE Attribute

The LIKE attribute has the following form:

LIKE lr

where lr is the like reference. The like reference must be a name or a sequence of names separated by periods. A use of the LIKE attribute must satisfy the following restrictions:

- The LIKE attribute can be used only in a DECLARE statement.
- The LIKE attribute can apply only to a structure name; that is, it must appear in a declaration clause that begins with a level number.
- A structure declaration with a LIKE attribute must not be followed by a member declaration. That is, if the declaration clause that contains the LIKE attribute has level number n, then the immediately following declaration clause must not have a level number that is greater than n.
- It must be possible to resolve the like reference in a LIKE attribute according to the rules for the resolution of name references given later in this section.
- The structure designated by the like reference must not contain a LIKE attribute. That is, a LIKE attribute cannot be defined in terms of some other LIKE attribute.

The Interpretation of the LIKE Attribute

A LIKE attribute is interpreted by the compiler. First, the like reference is resolved; the result is the declaration of a structure. The designated declaration consists of a declaration clause for the structure name itself followed by a sequence of declaration clauses for the members of the structure, the members of the members of the structure, and so on. The compiler copies the sequence of member declaration clauses into a position immediately after the declaration clause that contains the given LIKE attribute; then it deletes the LIKE attribute. The final result is a complete declaration of a structure.

The designated declaration clauses are copied literally, before any attributes are filled in by the various default mechanisms of PL/I. The level numbers are adjusted, if necessary, to assure that the declaration clause for a member has a higher number than that for the containing structure. Within a given block, the result of interpreting one LIKE attribute is not used in interpreting another LIKE attribute; such possibilities are ignored when a like reference is resolved.

Examples of the LIKE Attribute

The following program contains three examples of the use of the LIKE attribute:

```
P:   PROC;
        DCL 01 CUSTOMER(1000) EXTERNAL CONTROLLED,
               02 IDENT,
                  03 NAME(3) CHAR(30),
                  03 NUMBER PIC"999B99B9999",
               02 BALANCE DEC(8,2);
        DCL 01 CURRENT BASED LIKE CUSTOMER;
        DCL 01 IDENT_LIST(20) LIKE CUSTOMER.IDENT;
        DCL 01 IDENT_PAIR EXTERNAL STATIC,
               02 OLD LIKE CUSTOMER.IDENT,
               02 NEW LIKE CUSTOMER.IDENT;

        ...
        END;
```

The last three DECLARE statements are interpreted as follows:

```
DCL 01 CURRENT BASED,
       02 IDENT,
          03 NAME(3) CHAR(30),
          03 NUMBER PIC"999B99B9999",
       02 BALANCE DEC(8,2);
DCL 01 IDENT_LIST(20),
       02 NAME(3) CHAR(30),
       02 NUMBER PIC"999B99B9999";
DCL 01 IDENT_PAIR EXTERNAL STATIC,
       02 OLD,
          03 NAME(3) CHAR(30),
          03 NUMBER PIC"999B99B9999",
       02 NEW,
          03 NAME(3) CHAR(30),
          03 NUMBER PIC"999B99B9999";
```

Guidelines for the LIKE Attribute

A LIKE attribute should not be used merely to save writing; it should be used only when there is a close relationship between structure variables.

THE DEFAULT STATEMENT

A DEFAULT statement is a rule for adding attributes to the declaration of a name, a constant literal, a parameter, or the returned result of a procedure. The statement applies to every such declaration within its scope.

A DEFAULT statement is composed of a default test, which examines the attributes already present in a given declaration, and a sequence of default attributes that are added to the given declaration when the default test is satisfied. A DEFAULT statement is fully interpreted by the compiler, and it has no direct action when a program is executed.

The definition of the DEFAULT statement is not given in this manual; it appears in the PL/I Language Manual. A few examples are given here, however, in order to provide a brief introduction to the statement.

Examples of the DEFAULT Statement

As an example of the use of a DEFAULT statement, suppose it is necessary to use double precision for binary floating-point values in a certain block. The following statement can be used to achieve the desired effect by introducing a new default for the number-of-digits:

```
DEFAULT(FLOAT & ^DEC & ^PREC) PREC(63);
```

This statement means "wherever a FLOAT attribute is present and a DECIMAL attribute is not present and a PRECISION attribute is not present, insert the PRECISION(63) attribute".

The example just given requires some discussion. Why not use BIN instead of ^DEC? And why use ^PREC at all? The answer is that a default test must be written very carefully to cover all possible cases. Consider the following statement:

```
DCL X FLOAT;
```

Even though the system defaults will eventually add BINARY to this declaration, they are applied after, not before, the DEFAULT statement is applied. Therefore, the use of BIN in the default test instead of ^DEC would miss this declaration of X. Next, consider:

```
DCL Y FLOAT PREC(30);
```

A DEFAULT statement adds an attribute rather than replacing an attribute. If the ^PREC is omitted from the default test, then the DEFAULT statement applies to this declaration of Y and the result is:

```
DCL Y FLOAT PREC(30) PREC(63);
```

This is an invalid DECLARE statement.

A DEFAULT statement can be used to exclude certain declarations; for example:

```
DEFAULT(COMPLEX) ERROR;
```

This statement means "wherever a COMPLEX attribute is present, the program is in error". When the default test is satisfied for this DEFAULT statement, the compiler prints a diagnostic message. This example suggests that the DEFAULT statement could be used to enforce the use of a subset of PL/I. However, the DEFAULT statement is too limited to cover many such cases. For example, there is no way to require that a FIXED BINARY value have a zero scale factor.

As a third example of a DEFAULT statement, consider the following:

```
DEFAULT(^(RANGE(I:N))
      & ^(CONSTANT!BUILTIN!CONDITION!GENERIC)
      & ^(CHARACTER!BIT!POINTER!OFFSET!AREA!LABEL!ENTRY!FILE)
      & ^(FIXED!DECIMAL!PRECISION))
      FLOAT BINARY PREC(27);
```

This statement is complicated because it must cover all possible cases; however, it means that an arithmetic variable that does not begin with IJKLMN is assumed to be FLOAT BINARY PRECISION(27) when attributes to the contrary are not given. When used in conjunction with the system defaults, it approximates the handling of variable names in FORTRAN.

Guidelines for the DEFAULT Statement

The examples of the DEFAULT statement just given show that it is difficult to use. The problems with the DEFAULT statement can be summarized as follows:

- It is too deep. The DEFAULT statement is applied after certain special defaults are applied and before the standard system defaults are applied; furthermore, the order in which several DEFAULT statements are written can affect their results.
- It is too limited. The test in a DEFAULT statement is not powerful. Many useful defaults cannot be programmed and others can be programmed only in an indirect and complicated way.
- It is unnecessary. PL/I already has an elaborate set of standard system defaults. A departure from those defaults introduces unwelcome complications.

For these reasons, the use of the DEFAULT statement is not recommended.

THE RESOLUTION OF NAME REFERENCES

This discussion of the resolution of name references begins with an example; then the specific definitions and rules for resolution are given.

A Preliminary Example of the Resolution of Name References

For examples of the resolution of names, consider once again the following program, which was given at the beginning of this section:

```
P:    PROC;
         DCL X FLOAT;
         DCL (SYSIN,SYSPRINT) FILE;
L2:      GET LIST (X);
         CALL Q;
         IF X ^= 0 THEN GOTO L2; ELSE RETURN;
Q:       PROC;
              DCL L2 FLOAT;
              L2 = X**2 - 3*X**2;
              PUT SKIP LIST(X,L2);
              END;
      END;
```

Consider the five instances of the name L2 in this program. One instance is in a label prefix and another is in a DECLARE statement; these establish declarations for L2. Two instances of L2 remain, and these instances are name references that are resolved as follows:

- The name reference in the GOTO statement is resolved to the declaration of L2 that is established in the outer block; therefore, this name reference is associated with the outer block and has attributes LABEL INTERNAL CONSTANT.

- The name references in the assignment statement and the PUT statement are resolved to the declaration of L2 that is established in the inner procedure; therefore, each of these name references is associated with the inner block and has the attribute FLOAT.

The rules under which this resolution was performed are given later in this section. First, however, some definitions must be given.

The Name-Sequence Set for a Declaration

Each declaration has an associated set of name sequences. If the declaration describes a structure variable, then the set contains the level-one name of the structure and also contains each sequence of names that is formed by starting with the level-one name and proceeding through contained level names. If the declaration does not describe a structure variable, then the set contains just one name-sequence and that name sequence is the declared name.

Four examples of declarations and their associated sets of name sequences are given in the following table:

Declaration	Set of Name Sequences
DCL 01 Q(0:5) BASED, 02 R1 FLOAT, 02 R2 DEC(8,2);	Q Q.R1 Q.R2
DCL 01 S AUTOMATIC, 02 WEF(2,M-3), 03 P FLOAT, 03 Q, 04 RHO(N) DEC(4), 04 PHI FIXED, 03 R FLOAT, 02 G(10,10,CNT) FLOAT;	S S.WEF S.WEF.P S.WEF.Q S.WEF.Q.RHO S.WEF.Q.PHI S.WEF.R S.G
DCL ALPHA(N+2) FLOAT CONTROLLED;	ALPHA
DCL SQRT BUILTIN;	SQRT

The Name-Sequence for a Name Reference

Each name reference has an associated name sequence. If the name reference is a structure-qualified variable reference, then the associated name sequence is the sequence of level names in the reference. In all other cases, the name sequence is just the name itself.

Four examples of name references and their associated name sequences are given in the following table:

Name Reference	Name Sequence
Q(3*N1+N2).R1	Q.R1
S.WEF(1,M-3).Q	S.WEF.Q
ALPHA(2*I-BETA)	ALPHA
SQRT	SQRT

The full definition of structure-qualified variable references is given later, in the section on "Expressions".

The Applicability of Declarations

A declaration can be applicable to a given name reference in two ways, as follows:

- The declaration is applicable if it has a name sequence that is identical to the name sequence for the given name reference. In this case, the name reference is fully-qualified with respect to the declaration.

- The declaration is applicable if it has a name sequence that, after the omission of one or more names, becomes identical to the name sequence for the given name reference. If only this case applies, then the name reference is partially-qualified with respect to the declaration.

As a basis for some examples of applicability, consider the following declaration of the structure OMEGA:

```
DCL 01 OMEGA CONTROLLED,
       02 M,
          03 S1 CHAR(30),
          03 S2 FIXED,
       02 GAMMA FLOAT;
```

A complete list of the references to which this declaration is applicable follows:

Fully-Qualified	Partially-Qualified
OMEGA	M
OMEGA.M	OMEGA.S1
OMEGA.M.S1	M.S1
OMEGA.M.S2	S1
OMEGA.GAMMA	OMEGA.S2
	M.S2
	S2
	GAMMA

The Resolution Rules

To resolve a given name reference, begin by trying to find a block that contains the given reference and that has an established declaration that is applicable to the given use of the name reference. There are three possibilities, as follows:

- No such block is found. In this case, the name reference is undeclared. The use of an undeclared name reference is not necessarily an error in Standard PL/I; a declaration is supplied according to the rules for contextual and implicit declaration, mentioned earlier in this section. However, the use of an undeclared name reference is not recommended in GCOS PL/I and the compiler marks such a use with a warning.

- Exactly one such block is found. In this case, that is the desired block and resolution proceeds as in the next paragraph.

- More than one such block is found. In this case, the desired block is the smallest of the blocks, and resolution proceeds for that block according to the next paragraph.

When the desired block has been found, there are three possible cases to be considered, as follows:

- If the block contains exactly one applicable declaration, then that declaration is the declaration of the given name reference and resolution is complete.

- If the block contains more than one applicable declaration, but only one for which the given name reference is fully-qualified, then that one is the declaration of the name reference and the resolution is complete.

- If the block contains more than one applicable declaration but the preceding case does not apply, then the declaration is <u>ambiguous</u> and the name reference is invalid.

As a basis for examples of name resolution, consider the following program:

```
P:    PROC;
         DCL X FLOAT;
         DCL Y FLOAT;
         ... (Computation #1)
Q:       PROC;
              DCL 01 S,
                     02 Y FLOAT,
                     02 Z DEC(8,2);
              DCL 01 R,
                     02 ALPHA CHAR(16),
                     02 Z(100) FLOAT;
              DCL ALPHA FIXED;
              ... (Computation #2)
              END;
         END;
```

Now consider some of the references that could appear in Computation #1 or Computation #2:

X — In either Computation, this reference is a fully-qualified reference to the X declared by the first DECLARE statement in the outer block.

Y — In Computation #1, this reference is a fully-qualified reference to the Y declared in the second DECLARE statement in the outer block. In Computation #2, this reference is a partially-qualified reference to the first member of S declared in the first DECLARE statement in the inner block.

ALPHA — In Computation #1, this reference is undeclared and, in GCOS PL/I, is contrary to recommended usage. In Computation #2, this reference is a partially-qualified reference to the first member of R and a fully-qualified reference to the ALPHA that is declared in the last DECLARE statement in the inner block. The reference is resolved to the second possibility, the ALPHA declared in the last DECLARE statement.

Z(I) — In Computation #1, this reference is undeclared. In Computation #2, this reference is a partially-qualified reference to the second member of both S and R. Both S and R are declared in the inner block; therefore, the reference cannot be resolved and is invalid.

Observe that the subscript in the reference Z(I) does not enter into the resolution of the reference, even though R.Z can have a subscript and S.Z cannot.

ATTRIBUTES

PL/I has more than 50 attributes, and many combinations of these can be used in the declaration of a name. The following paragraphs present two views of attributes. First, five kinds of names are defined and the sets of attributes allowed for the declaration of each kind of name are given. Then a complete classification of the attributes is presented.

Complete Attribute Sets

There are five main kinds of names, as follows:

variable names
constant names
built-in function names
condition names
generic names

The different kinds of names vary widely in their importance. The variable names have a variety and flexibility that overshadows all other names. Constant names and built-in function names appear in most programs. Condition names also occur in most programs, but they are used in a restricted context. Generic names are extremely specialized and are rarely used.

The following paragraphs give the complete attribute sets for each kind of name. A set of attributes is a valid complete declaration for a name if and only if it is included in one of these sets. As an example, consider one of the complete attribute sets for a variable name:

```
REAL FLOAT BINARY PRECISION(27) ALIGNED AUTOMATIC INTERNAL VARIABLE
```

Because of the default rules of the language, this attribute set can be shortened to:

```
FLOAT
```

However, the following paragraphs do not mention such shortened forms. They give only complete attribute sets, before any abbreviations or defaults have been applied. In practice, once an appropriate attribute set has been determined, it is a relatively routine job to apply abbreviations and defaults to shorten it.

The complete attribute sets are given by means of diagrams. The diagrams use two special notations, as follows:

- Braces indicate a choice; they enclose two or more lines, any one of which is chosen in making up a specific attribute set.

- Brackets indicate an option; they enclose an item that can be either included or omitted in making up a specific complete attribute set.

Some of the identifiers in the diagrams are underlined and some are not. The underlined identifiers are terms that are defined in the text that follows the diagram. The non-underlined identifiers are attribute keywords.

A variable name designates storage for a value obtained from input or calculation. The complete attribute sets for a variable name are:

```
dt [DIMENSION( bp,... )] {ALIGNED  } sc [INITIAL( x,... )] VARIABLE
                         {UNALIGNED}
```

where dt is the data type, bp,... is a sequence of array-bound pairs separated by commas, sc is the scope and class, and x,... is a sequence of initial value expressions separated by commas.

The data type is one of the following sets of attributes:

```
{ {REAL   } {FIXED} {BINARY } {PRECISION(p,q)}
  {COMPLEX} {FLOAT} {DECIMAL} {PRECISION(p)  }

  PICTURE"ps" [REAL   ]
              [COMPLEX]

  {CHARACTER( ee )} {NONVARYING}
  {BIT( ee )      } {VARYING   }

  LABEL [LOCAL]
  ENTRY [( d,... )] [OPTIONS(opt)] [RETURNS( d )]
  FORMAT [LOCAL]
  POINTER
  OFFSET [( a )]
  FILE
  AREA [( ee )]
  STRUCTURE [LIKE r]
}
```

where p is an unsigned decimal integer constant, q is an optionally signed decimal integer constant, ps is a picture, ee is an expression whose value can be converted to an integer value, d is a descriptor, opt is an OPTIONS type for external entries as described in the section "Procedure Invocation", a is a reference that yields an AREA value, and r is a like-reference.

The scope and class is one of the following sets of attributes:

```
{ { STATIC     } { INTERNAL }                                                  }
{ { CONTROLLED } { EXTERNAL }                                                  }
{                                                                              }
{ { AUTOMATIC                                                    }             }
{ { BASED [( lq )]                                               }             }
{ { PARAMETER                                                    } INTERNAL    }
{ { DEFINED( br )   [POSITION( i )]                              }             }
{ { MEMBER                                                       }             }
```

where lq is a locator qualifier, br is a based reference, and i is an expression whose value can be converted to an integer value.

Variable names are described in the section on "Expressions".

CONSTANT NAMES

A constant name designates a statement address or a file-state-block address. The address is set by the compiler and does not change during program execution. The complete attribute sets for a constant name are:

```
{ LABEL  INTERNAL  [DIMENSION( bp )]                                                                                  }
{                                                                                                                     }
{ ENTRY [(d,...)] { INTERNAL } [OPTIONS(opt)]  [RETURNS(d)]                                                           }
{                 { EXTERNAL }                                                                                        }
{                                                                                                                     } CONSTANT
{ FORMAT  INTERNAL                                                                                                    }
{                                                                                                                     }
{ FILE { INTERNAL } fd                                                                                                }
{      { EXTERNAL }                                                                                                   }
```

where bp is a pair of array bounds, d is a descriptor for a parameter or a result, opt is an OPTIONS type for external entries, and fd is the file description, defined as follows:

```
{ STREAM { INPUT                                                                  }                                  }
{        { OUTPUT  [PRINT]  [ENVIRONMENT( { CONSECUTIVE } )]                      }                                  }
{                                         { INTERACTIVE }                         }                                  }
{                                                                                 }                                  }
{ RECORD { INPUT  } { SEQUENTIAL       }        ENVIRONMENT ( { CONSECUTIVE } )   }                                  }
{        { OUTPUT } { SEQUENTIAL KEYED }                     { INDEXED     }     }                                  }
{        { UPDATE } { DIRECT KEYED     }                     { REGIONAL    }     }                                  }
```

Constant names are described in the section on "Expressions".

BUILT-IN FUNCTION NAMES

A built-in function name designates an operation that is applied to arguments to produce a result. Each built-in function name designates an operation whose action is a fixed part of the definition of PL/I. The complete attribute set for a built-in function name is:

INTERNAL BUILTIN

Built-in function names are described in a general way in the section on "Expressions", and the individual built-in functions are defined in the section on "Operations".

CONDITION NAMES

A condition name designates an exceptional situation that can arise during program execution; it is used in programming a response to the given situation. The complete attribute set for a condition name is:

EXTERNAL CONDITION

Condition names are described in the section on "Condition Handling".

GENERIC NAMES

A generic name designates a set of rules for selecting a programmed entry name from a set of programmed entry names. The compiler follows the rules and replaces the generic name with one of the programmed entry names. The complete attribute set for a generic name is:

INTERNAL GENERIC(alt,...)

where 'alt,...' is a sequence of alternatives separated by commas. Generic names are described in the section on "Procedure Invocation".

The Classification of Attributes

It is useful to arrange the attributes in a single, hierarchical classification and thus establish a complete terminology for the discussion of attributes. The classification is as follows:

storage description
  storage type
    data type
      computational
        arithmetic
          mode: REAL COMPLEX
          scale: FIXED FLOAT
          base: BINARY DECIMAL
          precision: PRECISION(p,q)
        string
          string type: CHARACTER(ee) BIT(ee) PICTURE"ps"
          variability: VARYING NONVARYING
      non-computational
        address
          statement: LABEL ENTRY FORMAT
          data
            locator: POINTER OFFSET
            file: FILE
        area: AREA(ee)
    aggregate type
      array: DIMENSION(bp,...)
      structure: STRUCTURE MEMBER
    alignment: ALIGNED UNALIGNED
  management class
    storage class
      allocation: AUTOMATIC STATIC CONTROLLED BASED(lq)
      shabring: BASED(lq) DEFINED(br) POSITION(i) PARAMETER
    scope: INTERNAL EXTERNAL
    category: VARIABLE CONSTANT
      initial: INITIAL(x,...)
  usage description
    label and format: LOCAL
    entry: ENTRY(d,...) RETURNS(d,...) OPTIONS(opt)
    offset: OFFSET(a)
    file constant
      operation: INPUT OUTPUT UPDATE
      organization
        stream: STREAM PRINT ENVIRONMENT(INTERACTIVE) ENVIRONMENT(CONSECUTIVE)
        record: RECORD SEQUENTIAL DIRECT KEYED ENVIRONMENT(CONSECUTIVE)
  non-valued names: ENVIRONMENT(INDEXED) ENVIRONMENT(REGIONAL)
    compile-time: LIKE r GENERIC(alt,...)
    intrinsic names: BUILTIN CONDITION

Definitions for the underlined identifiers, p, q, ee, and so on, are not given here because they were given earlier, in the definitions of the complete attribute sets. Further information about a given attribute or class of attributes can be located through the index of this manual.

SECTION VII

STORAGE MANAGEMENT

Part of the cost of program execution is expended on the computing resource called storage. When the storage requirement can be reduced, the cost of the program execution decreases. A programmer cannot do much to reduce the storage occupied by a given program, but he can exercise some control over the amount of storage required for the data on which the program operates. The control of data storage is called storage management. Observe that storage management concerns storage itself, and not the values contained in that storage.

Each variable name in a program must have storage allocated for its value at some time before it is assigned a value and may have that storage freed at some time after the last use of its value. Allocation and freeing are the fundamental operations of storage management. There are several mechanisms that cause these operations to occur, and the programmer chooses one mechanism for each variable name by giving a storage class attribute when he declares the variable name.

The requirement of a program for storage can be substantially reduced by the vigorous practice of storage management; the effect is achieved by using a given portion of storage for more than one purpose and thus "recycling" the storage resource. The resulting saving in storage cost is desirable; however, storage management has its own cost because it is, in itself, a form of data processing. In some cases, the saving of storage does not justify the increase in processing cost. Therefore, programmed storage management is usually reserved for cases in which a large saving in storage can be accomplished in a simple way.

The block structure of PL/I allows the programmer to organize programs in a way that implies the desired storage management without requiring explicitly programmed allocation and freeing of storage. In addition, PL/I has built-in routines of great sophistication that facilitate storage management under program control. These features of PL/I allow the programmer to use very powerful storage management techniques with a small programming effort.

This section has three main parts. The first part gives examples and the fundamentals of storage allocation. The second part defines the management class, which consists of the usage, scope, storage class, and initial attributes. The final part discusses the capacity of storage and the exceptional conditions that apply to storage allocation.

PRELIMINARY EXAMPLES OF STORAGE MANAGEMENT

A few concrete examples are given here to prepare for the detailed discussion of storage management. Consider, first, a program that has a minimum of storage management:

```
P1:  PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL (A,B,C) FLOAT STATIC;
        GET LIST(A);
        B = A+4;
        C = B**2;
        PUT LIST(B,C);
        END;
```

Because the variable names A, B, and C are declared STATIC, their storage is allocated and paid for throughout program execution. Since there are times when a variable is not in use (that is, does not contain a useful value), storage is wasted.

One way of introducing storage management is to use CONTROLLED variables. Then statements can be inserted that allocate each variable just before its first use and free it just after its last use, as in the following revision of the given example:

```
P2:  PROC;                                  0
        DCL (SYSIN,SYSPRINT) FILE;          0
        DCL (A,B,C) FLOAT CONTROLLED;       0
        ALLOCATE A;                         1
        GET LIST(A);                        1
        ALLOCATE B;                         2
        B = A+4;                            2
        FREE A;                             1
        ALLOCATE C;                         2
        C = B**2;                           2
        PUT LIST(B,C);                      2
        FREE B,C;                           0
        END;
```

The digit at the end of each line is not part of the program; it is added to show how many FLOAT variables are in storage after each statement. Instead of using <u>three</u> variables all the time, the program never uses more than <u>two</u>. On the other hand, the program has more statements, and these statements add to the cost of executing the program. The version just given would not be used to save one or two FLOAT variables; but it might be appropriate if large arrays, for example, were involved.

An entirely different approach can be taken to reducing the storage required for the given program. The programmer can recognize that the last use of the variable A occurs before the first use of the variable C; this, of course, is a special property of this particular calculation. The programmer could then replace each occurrence of C by A throughout his program and omit the declaration of C; but this would confuse the logic of the program (since A and C represent different mathematical quantities). A different approach is to use a DEFINED variable, as follows:

```
P3:  PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL (A,B) FLOAT STATIC;
        DCL C FLOAT DEFINED(A);
        GET LIST(A);
        B = A+4;
        C = B**2;
        PUT LIST(B,C);
        END;
```

The DEFINED attribute causes C to occupy the same storage as A, so the program needs storage only for two FLOAT variables. This storage management technique has virtually no cost in terms of execution; however, it requires an analysis of the program, and an error in that analysis can easily lead to costs far in excess of the saving in storage.

Both of these examples of programmed storage management have weaknesses: the first incurs a considerable processing cost for allocation and freeing and the second borders on trick programming. Such techniques are more appropriate when PL/I is used to program a computer with a small storage capacity. There are occasions when programmed storage management is necessary in PL/I programs; but most PL/I programs rely entirely on the allocation and freeing performed by PL/I in response to the structure of the program.

FUNDAMENTALS OF STORAGE MANAGEMENT

The PL/I processor assigns or arranges for the assignment of one or more main memory storage units for each variable declared in the program. The storage is assigned either before execution begins or in the course of execution, depending on the PL/I storage class attribute specified or assumed for the variable.

Storage Regions

The storage assignments can be divided into three types referred to here as storage regions.

A region is either internal or external. Each internal region is associated with a particular block of a program. There are two kinds of internal regions:

- A permanent internal region is used for variables with INTERNAL STATIC or INTERNAL CONTROLLED attributes. There is one such region for each block in a program. The region is created at some time before program execution begins and is not destroyed until after execution ends.

- An activation internal region is used for variables with INTERNAL AUTOMATIC attributes. There is one such region for each activation of each block in a program. The region is created when the block is activated and is destroyed when the block is deactivated.

An external region is associated with an entire program rather than a particular block. There is one external region:

- The ordinary external region is used for variables and constants that have the attribute EXTERNAL. There is one such region for an entire program. The region is created at some time before program execution begins and is not destroyed until after execution ends.

To summarize, the regions are classified according to the following hierarchy:

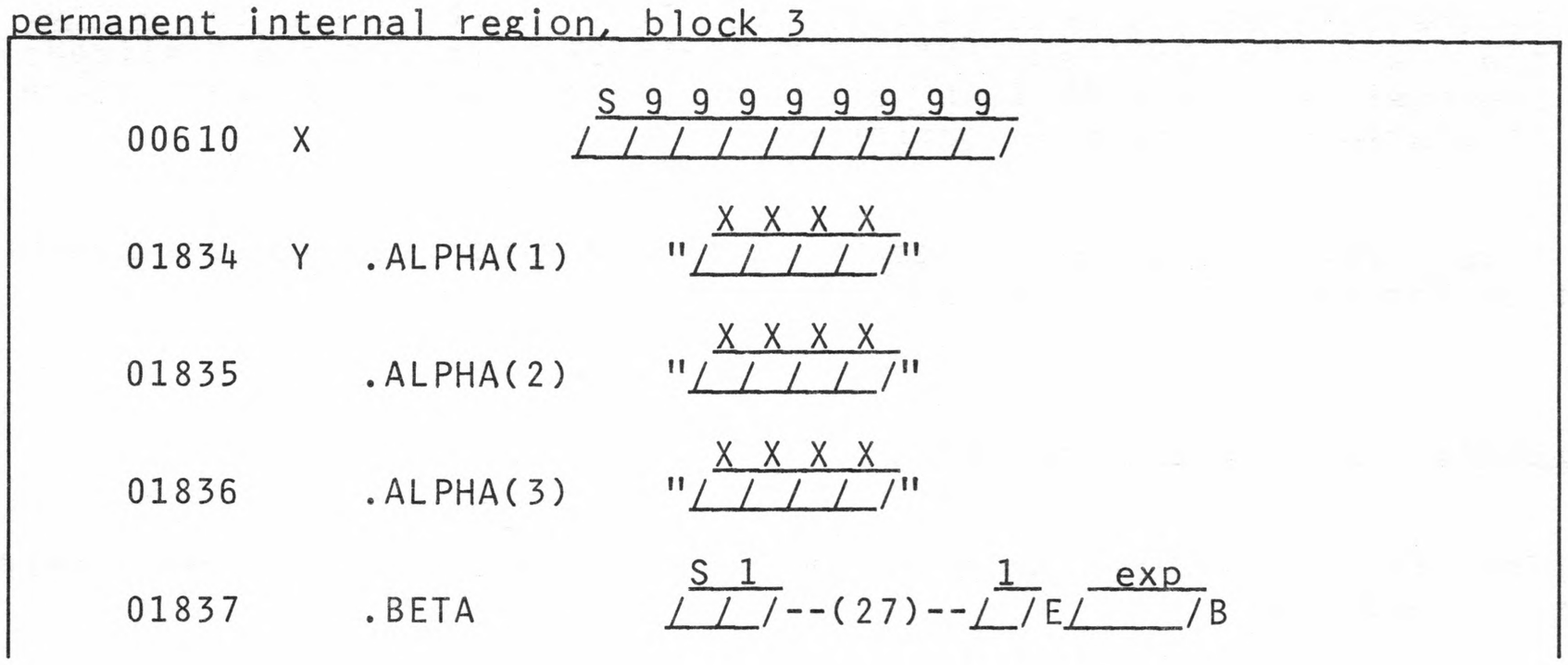
```
region
   internal
      permanent:  one per block
      activation: one per activation of each block
   external
      ordinary:   one per program
```

Region Diagrams

For purposes of discussion, it is useful to represent a region by a diagram. Suppose the following statements are all of the declarations in the third block of some given program:

```
BEGIN;
     DCL X FIXED DEC(8) STATIC INTERNAL;
     DCL 01 Y STATIC INTERNAL,
            02 ALPHA(3) CHAR(4),
            02 BETA FLOAT;
     ...
     END;
```

The program in question has a set of regions associated with it. The diagram of one of these regions is:



This region is the permanent internal region for the third block of the program. It contains five storage units, one for the scalar variable X and four more for a structure variable Y. The region continues downward, as suggested by the absence of a bottom edge for the enclosing boundary, and has room for many more storage units.

Each line of the region describes a storage unit. A line is made up of four components: a representation of a pointer value, a name, an access path composed of member names and subscripts, and a data frame.

Storage Management Operations

At any time during the execution of a program, each bit of storage in a region is either allocated or free. Storage is allocated if a designation that can be used in program execution is associated with it; otherwise, it is free. Free storage is organized into a pool from which requests for storage are satisfied.

The purpose of a storage management operation is to control the association between a designator (most commonly, a variable name) and its storage. The allocation operation for a variable name can be invoked either by the PL/I processor or by a program statement; but in either case, the details of the operation are determined by the attributes given in the declaration of the name. The allocation operation proceeds as follows:

1. Each extent expression (array bound, maximum string length, or area size) in the storage type attributes is evaluated and converted to an integer value. This evaluation is performed each time the allocation operation is performed.

2. The amount of storage required is determined from the given storage type attributes and their evaluated extents.

3. The region in which allocation must occur is determined from the given storage class attributes. Storage is withdrawn from the free storage pool of that region and is associated with the given variable name.

4. If the variable is an AREA variable, it is initialized to <u>empty</u>. If the variable has an INITIAL attribute, it is initialized; any expressions in the INITIAL attribute are evaluated each time the allocation operation is performed.

The <u>free operation</u> takes storage back from a given storage designator and returns it to the pool of free storage.

A Large Example of Storage Management

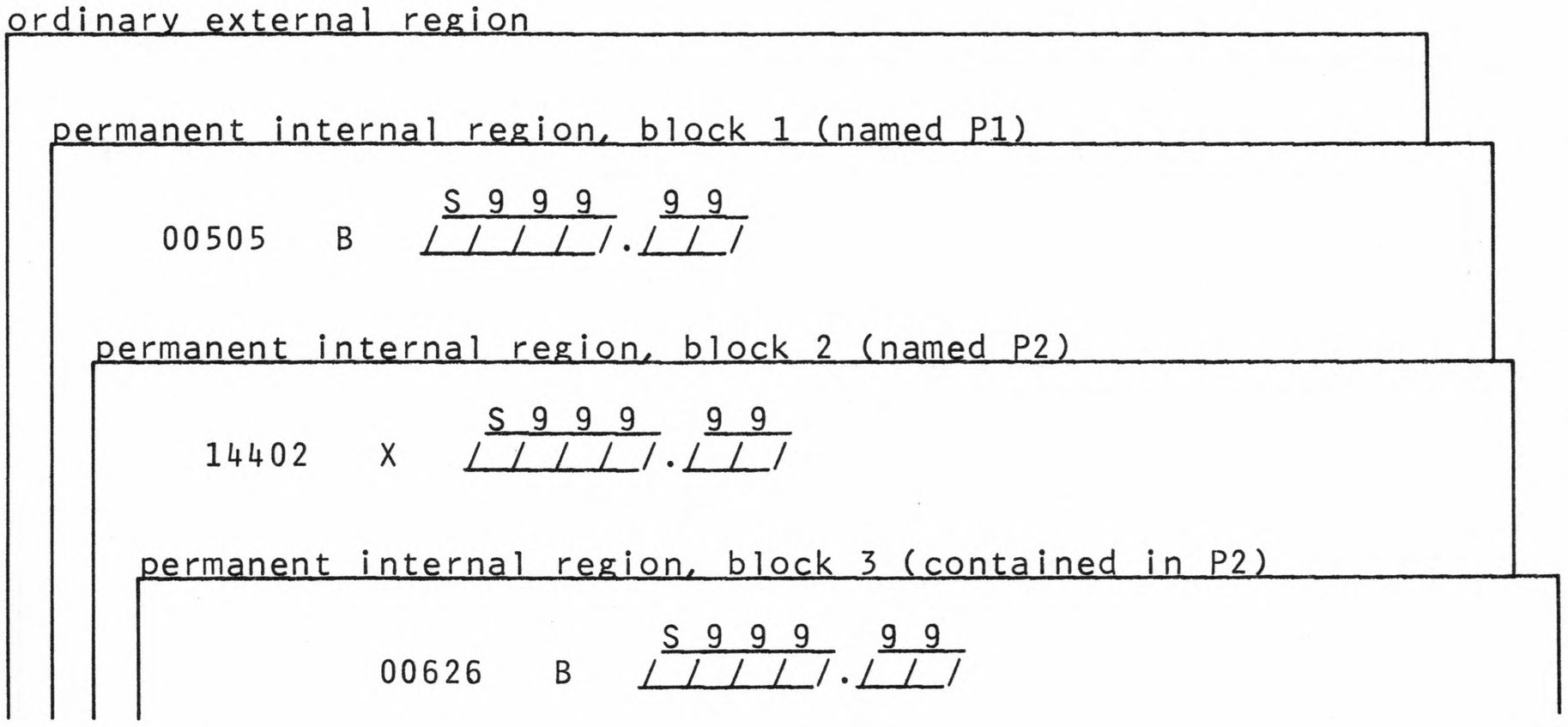
Consider the following program, which is made up of two external procedures, P1 and P2:

```
P1:  PROC;
        DCL P2 ENTRY EXTERNAL CONSTANT;
        DCL A DEC(5,2) CONTROLLED EXTERNAL;
        DCL B DEC(5,2) STATIC INTERNAL;
        ... (Computation #1)
        CALL P2;
        ... (Computation #2)
        FREE A;
        END;


P2:  PROC;
        DCL X DEC(5,2) STATIC INTERNAL;
        ... (Computation #3)
           BEGIN;
           DCL A DEC(5,2) CONTROLLED EXTERNAL;
           DCL B DEC(5,2) STATIC INTERNAL;
           DCL Y DEC(5,2) AUTOMATIC INTERNAL;
           ... (Computation #4)
           ALLOCATE A;
           ... (Computation #5)
           END;
     END;
```

The "Computations" in this program represent statements that do not introduce new declarations or affect the block structure of the program. Some of the attributes shown would normally be omitted by means of default conventions; to omit them here would not serve the purposes of an introductory example.

The following discussion traces the execution of this program and describes each allocation operation as it occurs. Before execution begins, a single ordinary external region is created; for this program, it is initially empty. Also before execution begins, a permanent internal region is created for each block and each STATIC INTERNAL variable is allocated. Immediately before execution of the program, the diagram for storage is:

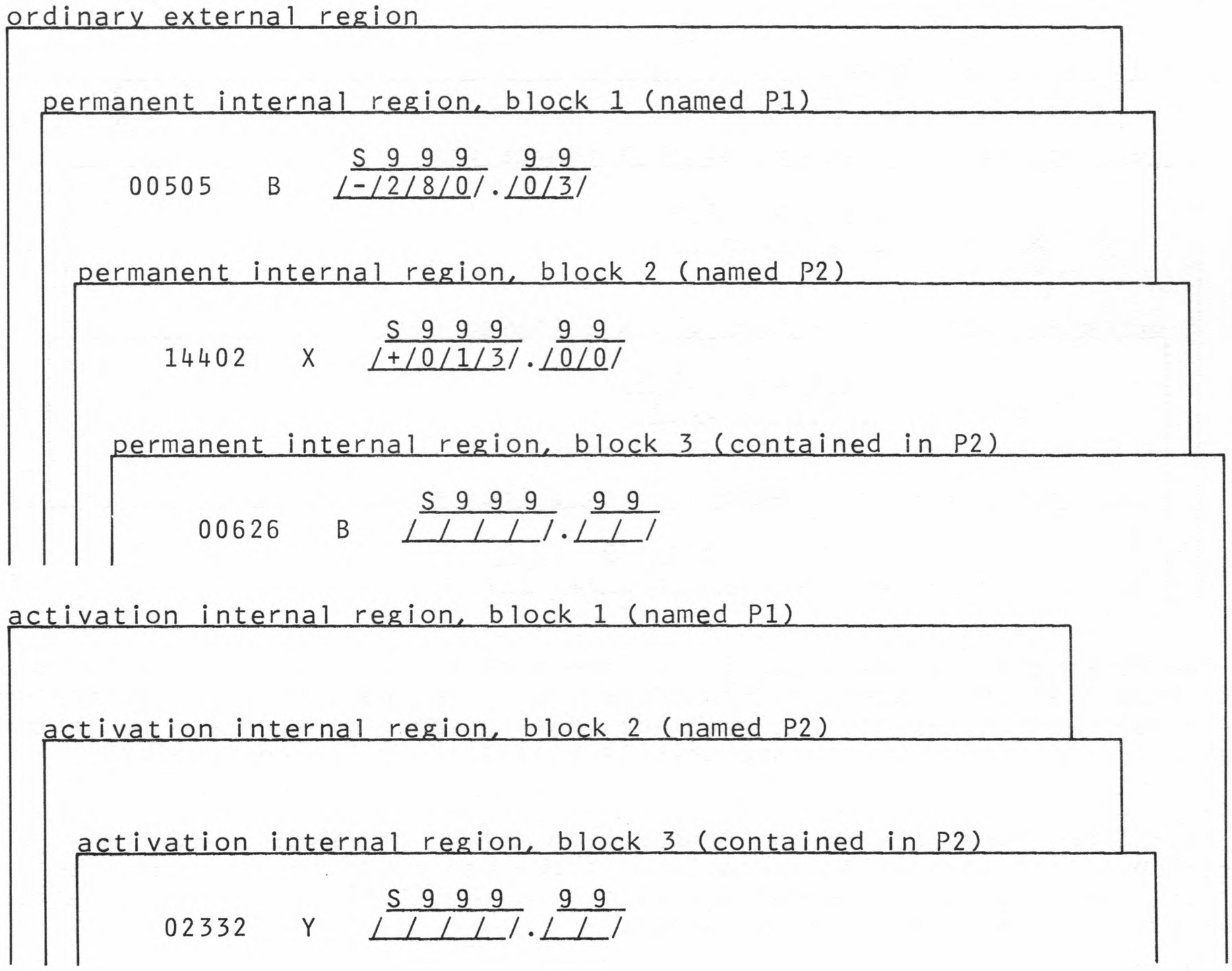


The first step in the execution of the program is the activation of the block that is the external procedure P1. As part of this activation, an internal activation region is created; but because there is no automatic variable in the block, no storage unit is allocated in the new region.

Execution continues with Computation #1. It is assumed that the only allocation of the controlled variable A is the one shown in P2; therefore, the only storage unit available for use so far is the static variable B. After the computation, the procedure P2 is called.

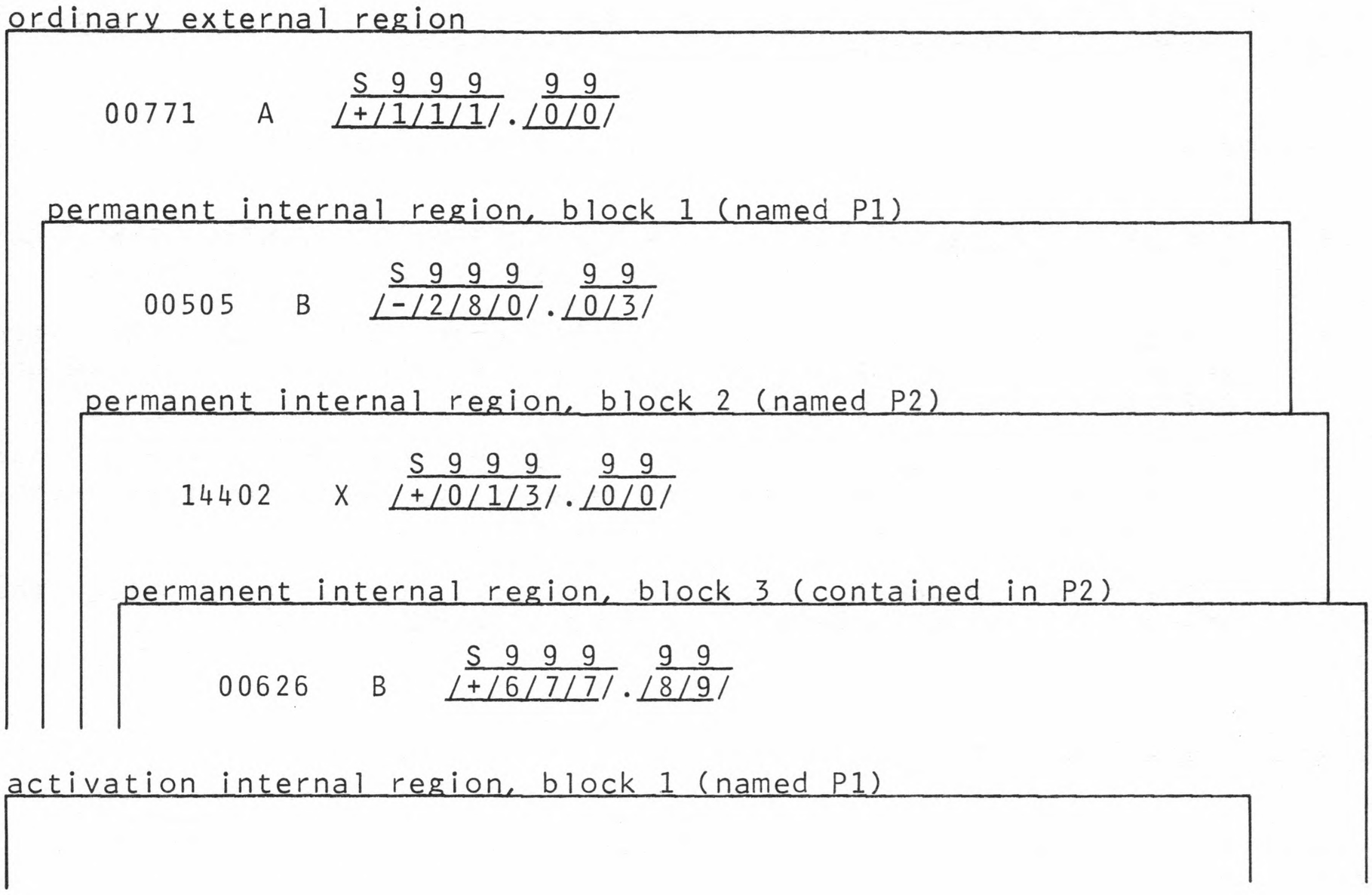
The first step in the execution of P2 is the activation of the block; an internal activation region is created, but once again there is no automatic variable in the block, so no storage unit is allocated in the new region. Execution continues with Computation #3, which can use only the static variable X. After the computation, control flows into the BEGIN block.

The execution of the BEGIN block begins with its activation. Again, an internal activation region is created; but now there is an automatic variable, Y, in the block and it is allocated as part of the activation. As Computation #4 is about to begin, the diagram for storage is:



Computation #4 can use the static variable B and the automatic variable Y. Observe, however, that the B that is accessible is the one in the third permanent internal region, not the one in the first. Thus in this example, P2 cannot use any value that was computed in P1.

After Computation #4 is executed, a statement is executed that allocates a storage unit for the controlled variable A in the ordinary external region. During Computation #5, a value is stored in A, and this, it is assumed, is a useful result that P2 prepared for use by P1. As execution of the BEGIN block and P2 is completed, their regions are discarded; then control returns to P1. As Computation #2 is about to begin, the diagram for storage is:



Computation #2 can use the static variable B allocated for block 1; in addition, now that a storage unit has been allocated for A, this variable can be used in the computation.

After Computation #2, a FREE statement returns the storage for A to the free storage pool. In the storage diagram, the ordinary external region once again is empty. The last step in execution of the program is the deactivation of the block that is external procedure P1; as part of this step, the internal activation region is discarded. At this time, the storage diagram looks just as it did before program execution (see the first diagram of this series) except that the stored values are filled in. When the process of which the program is a part ends, the remaining regions are discarded, and no storage associated with the program remains.

THE MANAGEMENT CLASS

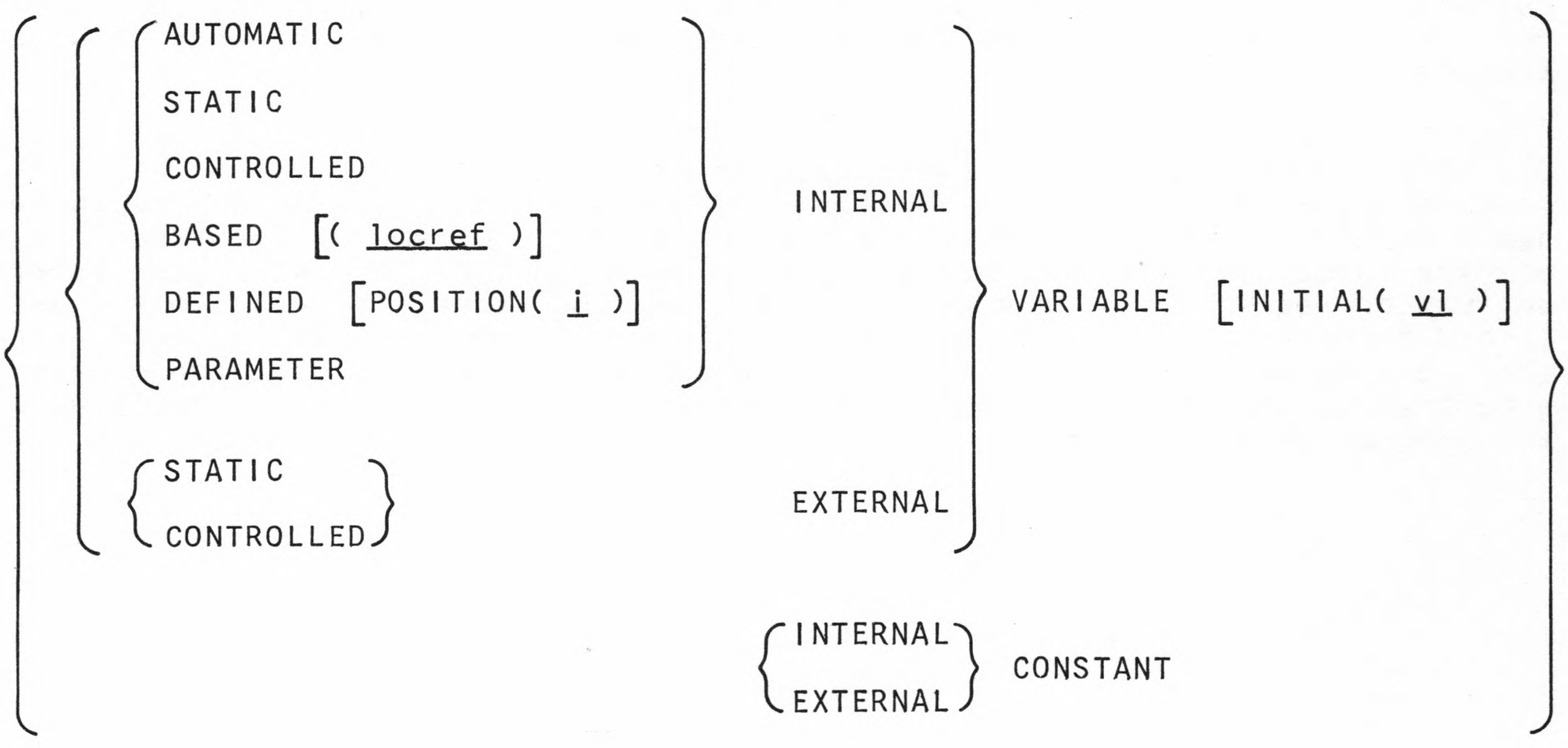
According to the earlier section on "Declarations", the declaration of a name is usually divided into two parts. The first part is the storage type, which specifies the kind of values accommodated by the designated variable or constant. The second part is the management class, which specifies various information about the handling of storage including, but not restricted to, the method of allocation. The storage type was defined earlier, in the section on "Value Storage". The management class is described here.

The management class is composed of four kinds of attributes, as follows:

- The usage category attribute is VARIABLE or CONSTANT. The attribute describes the way in which the storage is used by the program.
- The scope attribute is either INTERNAL or EXTERNAL. The attribute helps determine the region in which the storage is allocated and thus affects the accessibility of the storage.
- The class attribute is AUTOMATIC, STATIC, CONTROLLED, BASED, DEFINED, or PARAMETER. The attribute selects the mechanism that invokes the allocation and freeing operations.
- The initial value attribute, when present, specifies a value that is assigned to a variable when it is allocated.

The Form of the Management Class

Details of the form of the management class are given by the following diagram:



In this diagram,

- locref is a reference to a locative value or a function that yields a locative value.
- i is an expression whose value can be converted to an integer.
- vl is a list of initial value expressions for the INITIAL attribute and has the syntax given later in this section under "The INITIAL Attribute".

The INITIAL attribute must not be used with either the DEFINED or the PARAMETER attribute.

Abbreviations and Defaults

PL/I provides several abbreviations for the management class attributes, as follows:

Attribute	Abbreviation
INTERNAL	INT
EXTERNAL	EXT
AUTOMATIC	AUTO
CONTROLLED	CTL
PARAMETER	PARM
DEFINED	DEF
INITIAL	INIT

The default attributes for the category, scope and class are as follows:

Omitted Attribute	Default
category	default is VARIABLE exception: default is CONSTANT when the attribute ENTRY or FILE is present in the storage type and no attribute is present that cannot apply to a constant.
scope	default is INTERNAL exception: default is EXTERNAL when the attribute ENTRY or FILE is present in the storage type.
class	default is AUTOMATIC exception: default is STATIC when both EXTERNAL and VARIABLE attributes are present.

The following examples show the application of these defaults and abbreviations to typical attribute sets:

Full Allocation Type	Shortest Form
FLOAT AUTOMATIC INTERNAL VARIABLE	FLOAT
FLOAT STATIC EXTERNAL VARIABLE	FLOAT EXT
FILE EXTERNAL CONSTANT	FILE

REMARKS ON THE DEFAULT RULES

The default rules for the management class each choose the most commonly used attribute as the (non-exceptional) default: most names declared in DECLARE statements are indeed variables rather than constants; an INTERNAL name is used wherever possible because it is more economical; and AUTOMATIC variables are the hallmark of programming in a block-structured language like PL/I.

The exceptions for category and scope are not obvious and require further discussion. They reflect certain special patterns of PL/I programming and are best explained in terms of two specific DECLARE statements:

```
DCL P2 ENTRY EXTERNAL CONSTANT;
DCL SYSIN FILE EXTERNAL CONSTANT;
```

Declarations like these are the most common uses of the DECLARE statement for ENTRY or FILE names. Therefore, the default rules are adjusted so that these declarations can be given succinctly as:

```
DCL P2 ENTRY;
DCL SYSIN FILE;
```

The first statement is used when an external procedure (say, P1) calls another external procedure at an entry specified by P2. Since P2 does not appear in a label prefix in the procedure P1, it must be declared in that procedure by a DECLARE statement. The second statement is used to declare the name of a file constant. Something like it must occur in every program that performs input/output.

The exception for class is necessary because there is no AUTOMATIC class for an EXTERNAL variable. The exception uses STATIC as the default because it is simpler and more economical than CONTROLLED.

USAGE CATEGORIES

The usage category attributes indicate the ways in which names are used. For the VARIABLE category, storage management is explicit and is largely under the control of the programmer. For the CONSTANT category, the programmer can specify the scope attribute but must leave the details of allocation to the PL/I processor.

Variables

A variable storage unit sequence, or variable for short, is used to store a value that can change repeatedly throughout its existence and can be set directly by a program statement. A variable can be allocated and freed by the PL/I processor or explicitly by program statements; but in every case, the details of the operation are determined by the scope and class attributes that are given in the declaration of the variable name.

A variable can be declared with any storage type allowed in the language. A variable can accommodate a scalar value, in which case it is a single storage unit, or it can accommodate an aggregate value, in which case it is a sequence of storage units.

Constants

A constant storage unit sequence, or constant for short, is used to store a value that is set by the PL/I processor when storage is allocated and does not change thereafter. The only valid reference to a constant in a program is one that retrieves its value.

Computational constants are designated by literal constant references, while non-computational constants are designated by names. These program constructs are described later, in the section on "Expressions".

The design of PL/I does not allow the use of constants that can use a large quantity of storage. The value of a string constant cannot be longer than 64 words. No area constant is allowed. The only aggregate constant allowed is a one-dimensional array of label constants, which if used properly, should not be large. Thus the storage management of constants is of no interest to the programmer.

SCOPES

The scope attribute determines whether or not two declarations of the same name have the same meaning.

Internal Names

Each time a given name is declared in a different block with the INTERNAL attribute it has a different meaning. Each declaration can associate a different set of attributes with the name. When storage is allocated for the name, it will be different from that for other declarations of the name because it will be allocated in a region uniquely associated with the block in which the declaration is established.

External Names

Each time a given name is declared in a different block (but in the same program) with the EXTERNAL attribute it has the same meaning. Each declaration must associate exactly the same set of attributes with the name (after defaults and abbreviations have been filled in). Storage is allocated for the name only once, and is shared by all EXTERNAL declarations of the name because it is allocated in a region associated with the program as a whole.

Guidelines for the Scope

The INTERNAL attribute should be used except where there is a need to share a variable or constant between two external procedures. Rather than passing the variable or constant on each call from one external procedure to another, it can be declared external to reduce the time and space required for the call.

When external names are used, special care must be taken. If the external procedures of a large program are written at separate times or by different people, the use of external names must be coordinated. Otherwise, conflicting uses of the same external name will be discovered only when the complete program is executed, if at all.

The Correspondence Between Names and Storage

It is a principle of PL/I that each allocated portion of storage is designated by a name (that is, a declared identifier) in the program and, conversely, each name in the program designates an allocated portion of storage. Thus there is a correspondence between names and storage. This is a useful principle; but it does have some exceptions.

An allocated portion of storage is not designated by a name in the following cases:

- Temporary storage units are not designated by names (or any other program constructs). This is because temporaries are accessible to the PL/I processor but not to program statements.
- Computational constants are designated by literal constant references, not names. This is because the spelling of a literal constant is used to indicate the value of the constant. (For example, the literal constant 100 indicates that its value is one hundred.)
- Explicitly allocated based variables are designated by locative values, not names. More is said of this later in this section.

A name does not designate an allocated portion of storage in the following cases:

- A generic function name is used as a kind of macro name, and is replaced by an entry constant name (which does designate storage) before program execution begins.
- A built-in function name has an intrinsic meaning, and, in that respect, resembles an operator such as '+' or '='; for example, when ABS is declared BUILTIN, it refers to the absolute value operation that is part of the definition of PL/I.
- A condition name is related to an entry variable name; but the storage it uses is not directly accessible to program statements and is managed in a specialized way.
- A constant name is evaluated during program execution.
- A controlled variable name does not designate storage unless it is allocated.

Every name in a PL/I program has a scope. The scopes for the exceptional names just mentioned are INTERNAL for generic function references and built-in function names and EXTERNAL for condition names. These exceptional names are discussed in later sections.

STORAGE CLASSES

The storage class attribute determines storage management. By writing a single attribute for a variable name, the programmer selects from six very different mechanisms for storage management. Some mechanisms are easy to learn and use (AUTOMATIC, STATIC, and CONTROLLED). One is designed for the special purpose of transmitting arguments to procedures (PARAMETER). Those remaining (BASED and DEFINED) are for advanced programming applications and can be ignored in an initial study of PL/I.

Automatic Variables

The attribute AUTOMATIC designates a storage management mechanism that is driven by the block structure of the program rather than program statements. A variable whose name is declared AUTOMATIC is allocated as part of the activation of the block in which it is declared and is freed as part of the deactivation of that block. Thus it is available for use during a particular activation of the given block.

The activation of a block occurs as control enters a block (either by invocation of a procedure block or sequential flow into a BEGIN block); it is performed before the first executable statement of the block is executed. The deactivation occurs as control leaves the block (by execution of an END statement, a RETURN statement, or a GOTO statement); it is performed before execution of a statement that is not contained in the block.

An automatic variable always has INTERNAL scope. It is allocated in the activation internal region that is associated with the given activation of the given block.

VARIABLE EXPRESSIONS IN ATTRIBUTES FOR AUTOMATIC VARIABLES

The attributes of an automatic variable can contain variable expressions; these can be extent expressions (array bounds, maximum string length, or area size) or initial value expressions. When an automatic variable is allocated as part of block activation, these variable expressions must be evaluated. They must depend only on variables that are set before the given block activation begins.

This rule appears to be obvious, but the following program shows that it does introduce a restriction on the use of automatic variables:

```
P1:  PROC;
        DCL N FIXED INITIAL(5);
        DCL A(N) FLOAT;
        ...
        END;
```

(Since no storage class is given for N and A, both are AUTOMATIC.) Although it is quite clear what the programmer wants to have happen, this program is <u>invalid</u> because the allocation of A depends on a variable, N, that was not set before block activation began. In contrast, consider the program:

```
P2:  PROC;
        DCL N FIXED STATIC INITIAL(5);
        DCL A(N) FLOAT;
        ...
        END;
```

This program is valid because a STATIC variable is allocated and set before block activation begins.

THE SAVING OF VARIABLE EXTENTS FOR AUTOMATIC VARIABLES

The storage type used for every reference to a variable must be <u>consistent</u>; that is, it must be identical to the storage type used for the allocation of that variable. This is a fundamental rule of PL/I; it asserts that storage cannot be given more than one interpretation. There are exceptions to this rule, but they apply only when the storage class is BASED or DEFINED, and are relevant only in advanced applications of the language.

When BASED or DEFINED variables are not used, the design of PL/I makes it impossible for the programmer to break the rule of storage type consistency; specifically, when storage is allocated for a variable, every part of the storage type that can change during program execution is saved. Each reference to the variable uses the saved information and is therefore consistent with the allocation of the variable.

The following program fragment shows the effect of this treatment of storage types:

```
P1:  PROC;
        DCL N FIXED;
        N = 4;
        ...
           BEGIN;
           DCL S CHAR(N+2);
           ...
           N = 100;
           S = "ABCDEF";
           ...
           END;
        ...
        END;
```

(Since no storage class is given for N and S, both of the designated variables are AUTOMATIC.) At the time S is allocated, the storage type given by the DECLARE statement is CHAR(6); this storage type is used for the allocation and is saved. Subsequent reference to S, including both the assignment to S and the automatic freeing of S at the deactivation of the BEGIN block, use the saved storage type. Thus the fact that the storage type given by the DECLARE statement changes to become CHAR(102) has no effect on the program and, specifically, does not cause a violation of the storage type consistency rule.

The extents for an AUTOMATIC variable are saved in temporaries that are allocated and freed in the same region and at the same time as the variable. Because temporary storage is not accessible to program statements, the saved extents cannot be inadvertently changed.

Static Variables

The attribute STATIC designates a storage management mechanism that makes the allocated storage available throughout program execution. A variable whose name is declared STATIC is allocated at some time before the first reference to the variable is processed and is freed at some time after the last reference is processed.

An ordinary static variable can have INTERNAL or EXTERNAL scope and is allocated in a permanent internal region or in the ordinary external region, accordingly.

VARIABLE EXPRESSIONS IN ATTRIBUTES FOR STATIC VARIABLES

Every expression in an attribute in the declaration of a STATIC variable must be a constant expression. In particular, the extents (array bounds, maximum string length, or area size) and the initialization expressions must be constants. Since the extents are constants, it is not necessary to save them.

Controlled Variables

The attribute CONTROLLED designates a storage management mechanism that is driven by program statements and is thus under program control. A variable whose name is declared CONTROLLED is allocated by the execution of an ALLOCATE statement and freed by the execution of a FREE statement. Thus the variable is available for whatever portion of execution of the program the programmer requires.

A controlled variable can have INTERNAL or EXTERNAL scope and is allocated in a permanent internal region or in the ordinary external region, accordingly.

THE ALLOCATE AND FREE STATEMENTS FOR CONTROLLED VARIABLES

A controlled variable whose name is id is allocated by the statement

ALLOCATE id;

and is freed by the statement

FREE id;

The keyword ALLOCATE can be abbreviated as ALLOC. More than one identifier can be mentioned in an ALLOCATE or FREE statement; in that case, each identifier is separated from the next by a comma.

STACKING CONTROLLED VARIABLES

A simple use of a controlled variable name is to allocate storage for a variable, reference the storage as often as necessary, and then free the storage. A more general use of a controlled variable name is to allocate storage for a variable more than once before freeing its storage. When used in this way, a controlled variable name designates a stack of variables, and has the following properties:

- Each time the name is used in an ALLOCATE statement, a new variable is allocated and earlier allocations remain undisturbed.
- Each time the name is used in a FREE statement, the most recently allocated variable designated by the name is freed and the less recent allocations are undisturbed.
- Each time the name is used as a reference, the most recently allocated variable designated by the name is accessed for the setting or retrieving of a value.

A program that uses a name in a reference or a freeing when there is no variable allocated for that name is invalid. This can occur when no allocation has occurred or when the number of freeings already equals the number of allocations.

As an example of the use of a controlled variable name to designate a stack of variables, consider the following program:

```
P1:  PROC;
        DCL X FLOAT CONTROLLED;
        ...(Computation #1)
        ALLOCATE X;
        X = 10;
        ...(Computation #2)
        ALLOCATE X;
        X = 20;
        ...(Computation #3)
        FREE X;
        ...(Computation #4)
        FREE X;
        ...(Computation #5)
        END;
```

The "Computations" contain only references to X that retrieve its values; all allocations, freeings, and settings are shown explicitly. A reference to X in Computation #1 would be invalid because no variable has been allocated for X. References in Computations #2 and #3 yield 10 and 20. A reference in Computation #4 yields 10 because the variable first allocated for X becomes available after the first FREE statement; this is the point of the example. A reference to X in Computation #5 is invalid because all variables associated with X have been freed.

VARIABLE EXPRESSIONS IN ATTRIBUTES FOR CONTROLLED VARIABLES

The execution of an ALLOCATE statement causes the extent expressions (array bounds, maximum string length, or area size) and the initial value expressions in the declaration of the allocated variable name to be evaluated. The expressions can depend on any variable that has a value immediately before execution of the ALLOCATE statement.

Sometimes a variable expression in an attribute has access to different variables than the ALLOCATE statement that causes evaluation of the expression. Consider the program fragment:

```
P1:  PROC;
     DCL N FIXED;
     DCL A(N+2) FLOAT CONTROLLED;
     N = 1;
     ...
        BEGIN;
        DCL N FIXED;
        N = 10;
        ...
        ALLOCATE A;
        ...
        END;
     ...
     END;
```

The ALLOCATE statement uses the storage type DIM(3) FLOAT for the allocation of A, not DIM(12) FLOAT. The example emphasizes the rule, true throughout PL/I, that the resolution of a name depends on where it occurs in the program, not what it is used for.

THE SAVING OF VARIABLE EXTENTS FOR CONTROLLED VARIABLES

The values of any variable extents in the storage type of a CONTROLLED variable name are saved, as with AUTOMATIC variable names. The value saved and used for subsequent references is that computed when the variable is allocated; it is saved in a temporary that is associated with the variable. When more than one variable is allocated for a given controlled variable name, the saved extents may not be the same; they are evaluated for each allocation of the variable.

Parameter Variables

The attribute PARAMETER designates a storage management scheme that is an integral part of the invocation of a procedure, by either a CALL statement or a function reference. A variable name declared PARAMETER never has storage allocated for it; instead, it is associated with storage that has previously been allocated and that contains an argument for the procedure invocation. The parameter name is associated with the argument variable as part of the activation of the procedure block, and the association is broken as part of the deactivation of the block.

A parameter variable name always has INTERNAL scope. Because a parameter variable name is associated with a variable allocated for some other name, the storage it references can be in any region.

VARIABLE EXPRESSIONS IN ATTRIBUTES FOR PARAMETER VARIABLES

Each extent expression (array bound, maximum string length, or area size) must be either a constant expression or a single asterisk, '*'. When an extent expression is an asterisk, the corresponding extent of the associated argument variable is used. Full details of this mechanism, which is especially provided for parameters, is given later in the section on "Procedure Invocation".

A parameter variable never has storage allocated for it; therefore, it cannot have an initial attribute and the treatment of variable expressions in that attribute does not arise.

Based Variables

The attribute BASED designates a storage management mechanism that can be driven by program statements. A variable whose name is declared BASED can be allocated by an ALLOCATE statement and freed by a FREE statement. Such a variable is called an explicitly allocated based variable.

A based variable always has INTERNAL scope. An explicitly allocated based variable is allocated either in a permanent internal region or in an area variable; the choice depends on the ALLOCATE statement.

Whenever a variable is allocated for a based variable name, the storage type is supplied by the name but the name is not associated with the variable as its designator. Instead, the allocation operation produces a locative value that is assigned to a POINTER or OFFSET variable.

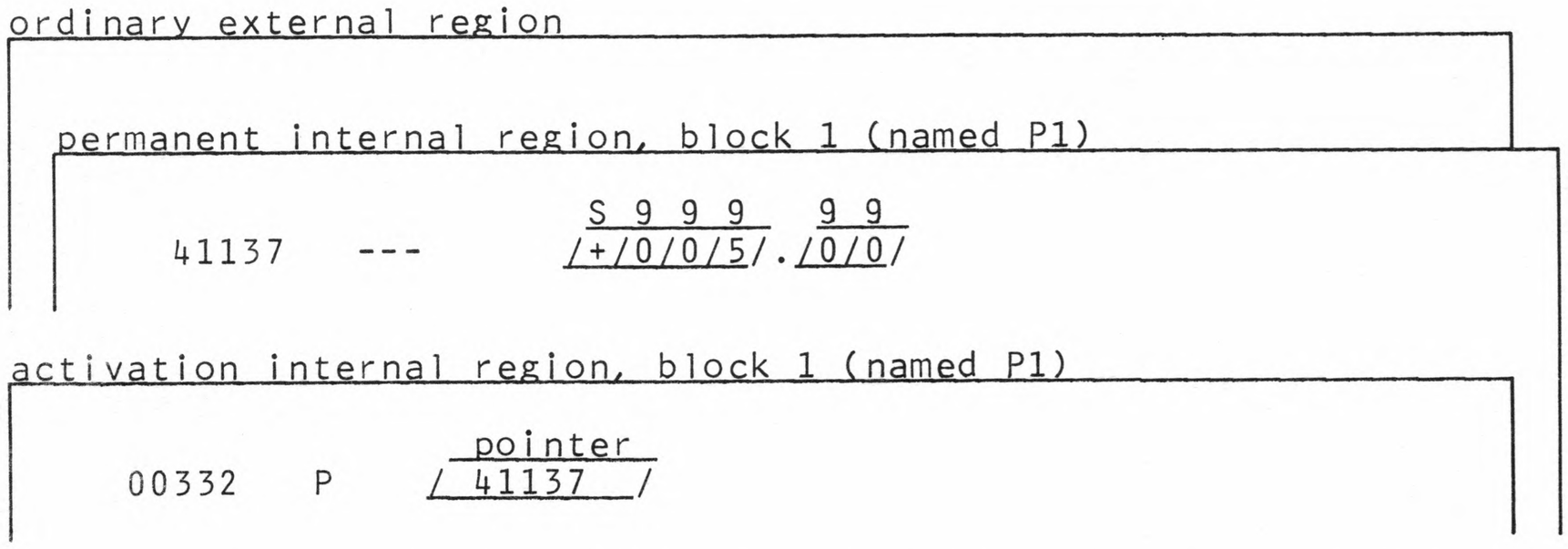
Whenever a variable is referenced with a based variable name, the storage type is supplied by the based variable name, but the name does not locate the variable in storage. Instead, the variable is designated by a locative value that is obtained by a POINTER or OFFSET variable that is mentioned with the based variable in the reference.

A simple example of the use of a based variable and an associated pointer is:

```
P1:  PROC;
        DCL A DEC(5,2) BASED INITIAL(0);
        DCL P POINTER;
        DCL SYSPRINT FILE;
        ...
        ALLOCATE A SET(P);
        ...
        P->A = 5;
        ...
        PUT LIST (P->A);
        ...
        FREE P->A;
        END;
```

In this example, the based variable name A is used in the allocation of an explicitly allocated based variable; but it is the value of the pointer P that designates that variable, not the based variable name A. Wherever A is used, the pointer variable name P is used with it.

As the PUT statement is about to be executed, the diagram for storage is:



Observe that the DEC(5,2) variable is not given the name A; instead, it must be designated by its locative value, represented as 41137.

The purpose of the based variable is not as self-evident as that of the other classes of variable. The discussion that follows gives the rules that govern the use of based variable, and illustrates those rules.

THE REFERENCE IN THE BASED ATTRIBUTE

The BASED attribute can be followed by a parenthesized reference to a locative value. This reference is adopted as the default locative reference for the variable name to which the attribute applies. If a locator qualifier is omitted from a reference to the based variable, the parenthesized reference is used as an assumed locative qualifier. Similarly, if the SET option is omitted from an ALLOCATE statement for the based variable, the parenthesized reference is used as an assumed SET option.

The example of a based variable given earlier can be shortened by the use of a parenthesized reference with the BASED attribute. The revised program is as follows:

```
P2:  PROC;
        DCL A DEC(5,2) BASED(P) INITIAL(0);
        DCL P POINTER;
        DCL SYSPRINT FILE;
        ...
        ALLOCATE A;
        ...
        A = 5;
        ...
        PUT LIST(A);
        ...
        FREE A;
        END;
```

Thus all mentions of the pointer P outside of the declarations are eliminated, and the usage for A closely resembles that of a controlled variable.

THE ALLOCATE AND FREE STATEMENTS FOR BASED VARIABLES

A based variable whose name is id is allocated by a statement of the form

ALLOCATE id [SET(locref)] [IN(arearef)] ;

and is freed by a statement of the form

FREE id [IN(arearef)] ;

The square brackets indicate that the options can be omitted. The locref in the SET option must be a reference to which a locative value can be assigned, namely a POINTER or OFFSET variable. The arearef in the IN option must be a reference that designates an AREA variable.

If id has been declared BASED(ref), then the SET option can be omitted and the PL/I processor assumes SET(ref). If the ref supplied by the BASED attribute is not a reference to which a locative value can be assigned, however, the assumed option will be invalid. When the ALLOCATE statement is executed, the locative value that designates the allocated storage is assigned to the variable supplied by the SET option or the BASED attribute.

If a programmer wants to allocate id in the internal permanent region, then the IN option is omitted. If a programmer wants to allocate id in some AREA variable (which can be in any region), then the area is specified by means of the IN option. When id is freed, the IN option must be the same that was used for its allocation.

VARIABLE EXPRESSIONS IN ATTRIBUTES

The execution of an ALLOCATE statement causes the extent expressions (array bounds, maximum string length, or area size) and the initial value expressions in the declaration of the allocated variable name to be evaluated. The expressions can depend on any variable that has a value immediately before execution of the ALLOCATE statement. In this respect, a based variable name behaves exactly as a controlled variable name.

THE SAVING OF VARIABLE EXTENTS

The values of variable extents in the storage type of a based variable name are <u>not</u> saved automatically by the PL/I processor; in this respect, based variables differ from all others. In some cases, the extents in the storage type are evaluated for each reference; and in these cases any change in an extent results in violation of the storage type consistency rule and an invalid program.

The following example is similar to a valid program fragment given earlier, in the description of automatic variables:

```
P1:  PROC;
        DCL N FIXED;
        DCL S CHAR(N+2) BASED(BETA);
        DCL BETA POINTER;
        ...
        N = 4;
        ALLOCATE S;
        ...
        N = 100;
        S = "ABCDEF";
        ...
        FREE S;
        END;
```

This program is <u>not</u> <u>valid</u>. The reference to S in the assignment statement causes the extent expression 'N+2' to be evaluated (since there is no stored extent available). Thus, although the variable was allocated with storage type CHAR(6), it is referenced with storage type CHAR(102). The string ABCDEF is filled out with blanks to make a string of 102 characters, and that string is assigned to S, where storage for only six characters has been provided. The result is the overwriting of variables allocated immediately after S. A similar error occurs when the storage for S is freed. These errors are not detected by the PL/I processor, and their effect may be an obscure malfunction of the program.

The invalid program just given can be repaired by keeping the expression for the maximum length of S unchanged throughout the existence of the variable for S. Toward this end, a new variable, N2 can be declared and used in the extent expression while the value of N is allowed to change as before. The revised program is:

```
P2:  PROC;
        DCL N FIXED;
        DCL N2 FIXED;
        DCL S CHAR(N2+2) BASED(BETA);
        DCL BETA POINTER;
        ...
        N = 4;
        N2 = N;
        ALLOCATE S;
        ...
        N = 100;
        S = "ABCDEF";
        ...
        FREE S;
        END;
```

This program is valid. It will evaluate the extent of the CHAR attribute three times (for the allocation, the reference, and the freeing), but it will obtain the value 6 each time.

THE REFER OPTION

PL/I has a special feature, the REFER option, that makes possible the systematic handling of variable extents of based variables. Using the REFER option, the program fragment considered earlier can be written as follows:

```
P3:  PROC;
        DCL N FIXED;
        DCL 01 SPAIR BASED(BETA),
               02 N2 FIXED,
               02 S  CHAR(N+2 REFER(N2));
        DCL BETA POINTER;
        ...
        N = 4;
        ALLOCATE SPAIR;
        ...
        N = 100;
        S = "ABCDEF";
        ...
        FREE SPAIR;
        END;
```

This revision of the original, invalid program has changed only the handling of S, replacing it with the structure SPAIR; however, the change eliminates the storage type inconsistency. The extent expression N+2 REFER(N2) is interpreted as follows:

> When SPAIR is allocated, calculate the extent for SPAIR.S (also known as S) from the expression N+2; use that value for the allocation of storage; and then save the value in SPAIR.N2 (also known as N2). Furthermore, whenever a reference to S is made, refer to N2, not the expression N+2', for the value of the extent.

Evidently, the REFER option has an important effect on the interpretation of the program.

The REFER option is used to save the variable extents of a based variable name in much the same way that the PL/I processor automatically saves variable extents for all other storage classes. Any number of extents in a variable can each be handled by a REFER option. For example, a structure that would otherwise be declared as

```
DCL 01 ALPHA(-2:R-2) BASED,
       02 BETA BIT(J(M-1)) VARYING,
       02 GAMMA(S1,S2) FLOAT;
```

can be declared as

```
DCL 01 X BASED,
       02 (K1,K2,K3,K4) FIXED,
       02 ALPHA(-2:R-2 REFER(K1)),
          03 BETA BIT(J(M-1) REFER(K2)) VARYING,
          03 GAMMA(S1 REFER(K3),S2 REFER(K4)) FLOAT;
```

Such a variable is called a self-defining structure because it carries its own extents within itself.

The parenthesized reference that follows the REFER keyword must designate a scalar variable that is declared earlier in the same structure that contains the REFER option. The referenced variable must, of course, be of suitable data type to have the extent value assigned to it.

EQUIVALENCED BASED VARIABLES

All of the based variables thus far discussed have been explicitly allocated based variables; that is, they are variables that were allocated using a based variable name to supply the storage type; and the locative value that designates the variable was produced as part of the allocation operation.

A second kind of based variable is the equivalenced based variable. Such a variable does not have storage of its own, but rather is superimposed on or equivalenced to a variable, the base variable, that was previously allocated. The base variable can have any storage type; it can even be an explicitly allocated based variable. However, the base variable must be connected; that is, if it is an aggregate, then its components must occupy an uninterrupted sequence of storage units.

The locative value that designates the base variable is obtained by the ADDR built-in function; for example, the assignment statement

```
P = ADDR(A);
```

obtains the locative value that designates the variable designated by A and assigns that locative value to the pointer variable designated by P.

There are three kinds of equivalenced based variables: simple based variables, string overlay based variables, and partial based variables; each of these is discussed in the following paragraphs.

Simple Based Variables

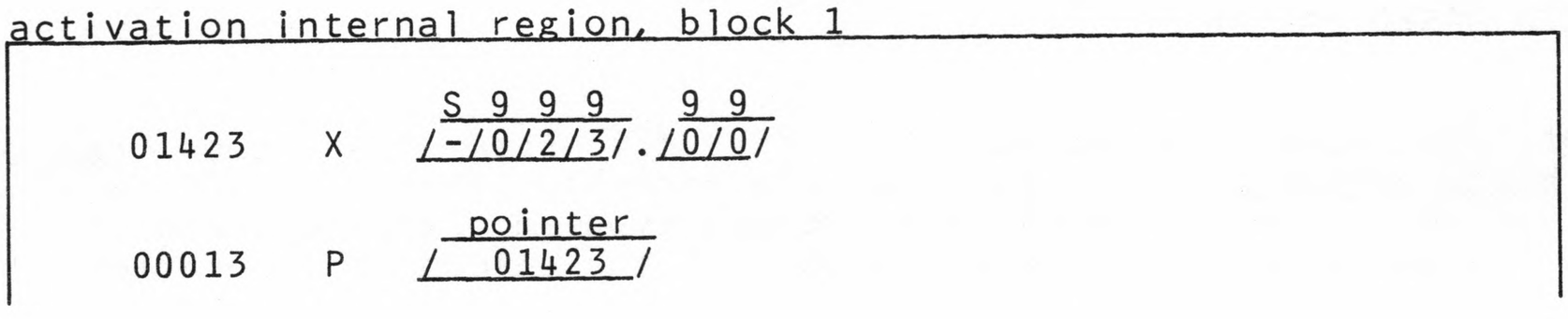
The storage type of a simple based variable must agree exactly with the storage type of the base variable.

As an example of the use of a simple based variable, consider the following program:

```
P1:  PROC;
        DCL X DEC(5,2);
        DCL Y DEC(5,2) BASED;
        DCL P POINTER;
        DCL (SYSIN,SYSPRINT) FILE;
        P = ADDR(X);
        GET LIST(X);
        PUT LIST(2*P->Y);
        END;
```

This program inputs a number, doubles it, and prints it. The program would be invalid if the storage type of Y differed from that of X.

Immediately after the GET statement has input the value -23 from the input stream, the diagram for storage is:



As is always the case with a based variable, there is no mention of Y in storage; it is used only to supply a storage type for use in conjunction with P.

As a second example of a simple based variable, consider the following statements:

```
DCL 01 A(5),
       02 X FIXED,
       02 Y CHAR(6);
DCL 01 B BASED,
       02 R FIXED,
       02 S CHAR(6);
DCL P2 POINTER;
```

Suppose that the following assignment statement is executed:

```
P2 = ADDR(A(3));
```

Then the following references are valid:

```
P2->B          (means A(3))

P2->B.R        (means A(3).X)

P2->B.S        (means A(3).Y)
```

Thus the POINTER variable supplies the address of the leftmost name, B in this example, in a reference to a based variable. Further examples are given later, in the section on "Expressions", under "Locator-Qualified Variable References".

String Overlay Based Variables

A string is not a truly indivisible value; it is made up of characters or bits. In some applications, it is useful to impose more than one interpretation on the storage for a sequence of characters or bits. String overlay based variables are used for this purpose.

A string overlay based variable and its base variable must be either NONVARYING UNALIGNED scalars or aggregates of NONVARYING UNALIGNED scalars. For a given string overlay, all variables must be CHARACTER or all variables must be BIT. The aggregate type of the base variable need not match, and a pictured character-string variable can be matched with an ordinary character-string variable. This relaxation of the rules for aggregate type and pictured data type permits different interpretations of a given string variable.

As an example of a string overlay based variable, consider the following statements:

```
DCL WORD(1024) BIT(36);
DCL 01 FLOAT_NUM BASED,
       02 SIGN BIT(1) UNAL,
       02 CHARACTERISTIC BIT(8) UNAL,
       02 MANTISSA BIT(27) UNAL;
DCL PNUM POINTER;
```

Suppose the following assignment statement is executed:

```
PNUM = ADDR(WORD(23));
```

then the following references are valid:

```
PNUM->FLOAT_NUM                  (means WORD(23))

PNUM->FLOAT_NUM.SIGN             (means the first bit of WORD(231))

PNUM->FLOAT_NUM.CHARACTERISTIC (means the next 8 bits of WORD(23))

PNUM->FLOAT_NUM.MANTISSA         (means the last 27 bits of WORD(23))
```

Thus the based structure variable can be used to designate various substrings of the string variable designated by WORD(23).

The requirement that strings be NONVARYING UNALIGNED is the way of saying, in PL/I, that several strings must be stored together, without any unused storage or any word or bit counts between them. Because the example satisfies this restriction, a single bit string can be interpreted as a structure made up of three bit strings.

Partial Based Variables

Sometimes it is necessary to examine a portion of a structure variable without knowing all the details of the storage type of the variable. A <u>partial based variable</u> is used for this purpose.

A partial based variable and its base variable must be structure variables whose storage types agree through the first <u>n</u> declaration clauses. For example, consider the following statements:

```
DCL 01 ALPHA,
       02 A FIXED,
       02 B,
          03 B1 FLOAT,
          03 B2 CHAR(10),
       02 C CHAR(2000) VAR;
DCL 01 BETA BASED,
       02 Q FIXED,
       02 R,
          03 R1 FLOAT,
          03 R2 CHAR(12),
       02 S CHAR(2000) VAR;
```

Observe that the first four declaration clauses of the storage types of ALPHA and BETA are the same; that is, both storage types begin:

```
01, 02 FIXED, 02, 03 FLOAT
```

Thus BETA can be used as a partial based variable name with ALPHA as the base variable name.

There are two restrictions on references to a partial based variable, as follows:

- When the storage types of the based variable name and the base variable name are compared from the beginning (the level-one declaration clause) toward the end (the last declaration clause), the storage types must agree through the declaration of all components that are designated by the given reference.

- Furthermore, if one of the components designated by the reference is a component of a level-two structure, then the declarations of the based variable and the base variable must agree for <u>all</u> components of that level-two structure.

The first restriction is a reasonable one; it requires that the portions of the structures that are used must be in agreement. The second restriction is not reasonable; it is a flaw in the design of PL/I.

Consider again the declarations of ALPHA and BETA given earlier in this discussion. Suppose the following assignment statement has been executed:

```
P = ADDR(ALPHA);
```

where P is declared POINTER. In this situation, the only valid use of BETA is the following reference:

```
P->BETA.Q        (means ALPHA.A)
```

Thus it is possible to obtain the value of ALPHA.A without matching the whole storage type of ALPHA.

Since the storage types of ALPHA and BETA agree through ALPHA.B.B1, it might seem valid to use BETA in the following reference:

```
P->BETA.R.R1
```

However, this reference violates the second restriction on partial based references. Specifically, the reference designates a component ALPHA.B.B1 of a level-two structure ALPHA.B that has a component ALPHA.B.B2 for which the storage type of ALPHA and BETA do not agree.

One use for a partial based variable reference arises in handling input from a file in which records have several different structures. If the file is designed so that every record begins with an integer value that indicates the structure of the remainder of the record, then this integer can be obtained by means of a partial based variable reference. An example is given later, in the section on "Record Input/Output" under "An Example of Based Input".

Defined Variables

The attribute DEFINED designates a storage management mechanism that provides a new name for an existing variable rather than allocating a new variable. The association between the defined variable name and the designated variable is formed at the time of block activation and broken at block deactivation.

A defined variable name always has INTERNAL scope. Because a defined name is associated with a variable allocated for some other name, the storage it references can be in any region.

VARIABLE EXPRESSIONS IN ATTRIBUTES FOR DEFINED VARIABLES

The extent expressions (array bounds, maximum string length, or area size) are evaluated as part of the activation of the block in which the defined variable is declared. Therefore, the expressions can depend only on variables that are defined before block activation begins.

A defined variable never has storage allocated for it; therefore, it cannot have an initial attribute and the treatment of variable expressions in that attribute does not arise.

THE SAVING OF VARIABLE EXTENTS FOR DEFINED VARIABLES

The values of the extent expression are saved in temporaries at the time they are computed. The storage for these temporaries is allocated in the activation region associated with the block activation, and the storage is freed upon block activation. Because the extents are saved, it is not possible to violate the storage consistency rule in the use of a defined variable.

THE USES OF DEFINED VARIABLES

The DEFINED attribute consists of the keyword DEFINED followed by a parenthesized reference. The variable designated by the reference is called the base variable because it is the base on which the defined variable is superimposed. There are three ways to use a defined variable: simple defining, string overlay defining and isub defining.

Simple Defining

For most cases of simple defining, the base variable must have the same storage type as the defined variable. Some examples follow; consider first:

```
DCL S CHAR(20) VARYING STATIC;
DCL T CHAR(20) VARYING DEFINED(S);
```

In this case, any reference to T will be equivalent to a reference to S. Consider next:

```
DCL 01 A,
       02 B(N),
          03 C FLOAT,
          03 D FLOAT,
       02 E CHAR(6);
DCL X FLOAT DEFINED(A.B(I-2).D);
DCL Y(N) FLOAT DEFINED(A.B(*).D);
DCL 01 Z DEFINED(A.B(J)),
       02 Z1 FLOAT,
       02 Z2 FLOAT;
```

In this case, X is associated with the scalar designated by A.B(I-2).D; the variable I is resolved where it appears (in the declaration) but is evaluated each time a reference to X is processed. The defined variable Y is associated with a cross-section of the array A.B, namely that formed of the second member of each element of the array. The defined variable Z is associated with a particular element of A.B; once again, J is evaluated for each reference.

String Overlay Defining

String overlay defining allows a string variable to be defined onto the storage of another string variable so that the defined variable occupies all or part of the same storage allocated for the base variable. String overlay defining can be performed for either bit strings or character-strings, but the two types cannot be mixed in any given use of string overlay defining.

The defined variable and the base variable must be NONVARYING UNALIGNED string scalars or aggregates of NONVARYING UNALIGNED string scalars. The aggregate type of the defined variable and the base variable need not match, and a pictured character-string variable can be treated as an ordinary NONVARYING character-string variable; these are the "loopholes" that make string overlay defining useful.

Examples of string overlay defining are given in the following DECLARE statements:

```
DCL A(5) CHAR(2) UNAL;
DCL B CHAR(8) DEF(A);
DCL 01 C DEF(A),
       02 X CHAR(5) UNAL,
       02 Y CHAR(5) UNAL;
```

The first statement declares A as the name of an array variable that has five elements, each composed of two characters. The second statement declares B as another name for the variable designated by A; but it views the variable not as an array but as a sequence of ten character positions, the first eight of which are interpreted as a scalar character-string variable. The third statement declares C as yet another name for the variable designated by A; but it interprets the character positions of A as a structure composed of two five-character members.

The POSITION attribute can be used to start the DEFINED variable at some position other than the first character position of the based variable. For example, consider:

```
DCL D CHAR(5) DEF(A) POS(6);
```

This statement declares a name for the last five character positions of the variable designated by A.

Isub Defining

For <u>isub defining</u>, the special lexemes 1SUB, 2SUB, 3SUB, and so on, are used to make special use of the subscript values of the defined variable. Consider the declarations:

```
DCL A(3,3) FLOAT;
DCL Q(3) FLOAT DEFINED A(1SUB,1SUB);
DCL R(3,3) FLOAT DEFINED A(4-1SUB,2SUB);
```

In this case, Q is made up of the three elements of A that lie on the diagonal, and R is made up of the rows of A in reverse order. The two arrays map onto A as follows:

```
Q(1) -- A(1,1)
Q(2) -- A(2,2)
Q(3) -- A(3,3)

R(1,1) -- A(3,1)   R(1,2) -- A(3,2)   R(1,3) -- A(3,3)
R(2,1) -- A(2,1)   R(2,2) -- A(2,2)   R(2,3) -- A(2,3)
R(3,1) -- A(1,1)   R(3,2) -- A(1,2)   R(3,3) -- A(1,3)
```

Guidelines for the Storage Class

Suggested uses for each of the six storage classes are given here. The order in which the storage classes are discussed is not an indication of their frequency of use.

AUTOMATIC VARIABLES

An automatic variable should be used wherever possible; the fact that AUTOMATIC is the default storage class attribute makes this recommendation easy to follow. An automatic variable is clearly associated with a block and is allocated only while that block is activated; therefore, its intended role in a program is more clear than that of any other class of variable. When a block is considered as a building block of a larger program, the automatic variable names declared in the block can be entirely ignored; their existence cannot be perceived from outside the block.

A reference to an automatic variable that is immediately contained in the block in which the variable name is declared is a local reference. Local references are considerably less costly to process than other references. In order to maximize the number of local references, an automatic variable should always be declared in the smallest existing block that contains all references to it; if this rule is followed, at least some of the references to the variable are local references.

If the programmer allocated a particular variable (controlled or based) at the beginning of execution of each block and freed it at the end of the block, the storage management operations would incur a considerable expense. Although an automatic variable is allocated and freed for each execution of its block, an extra cost is not incurred; on the contrary, the special techniques used in the implementation of PL/I (the use of a stack) make the cost of storage management of an automatic variable negligible.

The use of variable expressions in the extents of a storage type of an automatic variable increases the cost of the variable. Of course the extent expressions must be evaluated and saved when the variable is allocated; but this cost is obvious and is examined like any other computation cost. A more important cost arises in certain cases in references to the variable. Consider the declaration:

```
DCL 01 A,
       02 B(N) FLOAT,
       02 C FIXED;
```

For each reference to C, the PL/I processor must not only find the location at which the structure A begins, but also must determine the size of B so it can be skipped over to reach C. The size of B is not known at compile time, and therefore relatively inefficient code for the reference must be compiled. A more efficient declaration of the same structure is:

```
DCL 01 A,
       02 C FIXED,
       02 B(N) FLOAT;
```

When other considerations do not forbid, components with variable extents should be placed as late in a structure as possible.

Declaring an initial value for an automatic variable specifies that the initialization be performed each time storage is assigned the variable, that is, each time the block in which the variable is declared is activated. When the variable does indeed need to be reset to an initial value, this execution expense is reasonable, but when the value of the variable does not change from one activation to another, the initialization expense can be avoided by declaring the variable to be a static rather than an automatic variable.

STATIC VARIABLES

A static variable is used when a variable must exist outside the activation of the block in which the variable is declared. This requirement can arise in the following ways:

- A variable that is shared between several external procedures (compilable units) must be EXTERNAL and must therefore be STATIC or CONTROLLED. An external STATIC variable is less costly than an external CONTROLLED variable and therefore should be used where possible. Once again, the default conventions make this recommendation easy to follow.

- When a variable is used to keep information throughout the various invocations of a procedure, an INTERNAL STATIC variable should be used. Such a variable can be used, for example, to count the number of times a procedure is invoked.

CONTROLLED VARIABLES

A controlled variable is used when the built-in storage management mechanism of the automatic or static storage class does not suit a particular application. This requirement can arise in the following ways:

- When an external variable must have variable extents, a controlled external variable is a possible choice.
- When a stack of variables is needed, a controlled variable is a convenient choice. When a program is modified to be reentrant the static variables can be replaced by controlled variables; then each variable is allocated (pushed down) before each reentry and freed (popped up) after the reentry.
- When storage is a critical resource, controlled variables can be used to program a minimum use of storage.

PARAMETER VARIABLES

A parameter variable is used for the parameters of a procedure. For each parameter, execution time and space is required to transmit the location of the parameter from the calling to the called procedure, and in the called procedure, each reference to a parameter requires that this location be fetched. This execution expense can be reduced by declaring the would-be parameters as static variables in a block enclosing both procedures, or as external when the procedures are external to one another. Alternatively, the variables can be arranged in a structure or array, and the location of the structure or array can be provided as the value of an external variable or a transmitted parameter.

Any of these methods of improving execution efficiency, however, tend to obscure in what ways the two procedures are dependent upon one another. Using parameters makes explicit by means of what variables one procedure can affect the other.

BASED VARIABLES

A based variable is often used in a program that uses linked data structures. In such a program, locative values are used as the links and based variables are used for the structures connected by the links. The PL/I compiler, which is written in PL/I, makes extensive use of linked data structures and thus of based variables.

Only based variables can be allocated in an area variable; therefore based variables must be used in order to take advantage of the offset values and other special features that are available with area variables.

Based variables are also used in certain rather special ways. In some applications, it is necessary to operate on an aggregate value without fully knowing its storage type; in these cases, a based variable is used to examine part of the value. In string-processing applications that often arise in business programming, it is useful to interpret a given sequence of characters in more than one way by superimposing different aggregates of string variables; in these cases, based variables are used. These applications represent the situations in which PL/I makes gains in efficiency at the cost of breaking its rule of storage type consistency.

DEFINED VARIABLES

A defined variable is used to associate a new name with an existing variable. In certain cases, as when isubs are used with a defined array, the new name can be mapped onto the existing variable in a special way. There are occasions on which a defined variable can do what a based variable can do; and on these occasions the defined variable is preferred.

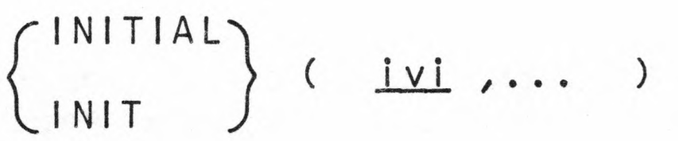
In practice, defined variables are not often used in PL/I. They are in competition with based variables, and the based variables are considerably more general. In PL/I programming, a defined variable should never be introduced for the sole purpose of using the same storage for two purposes and thus saving storage.

THE INITIAL ATTRIBUTE

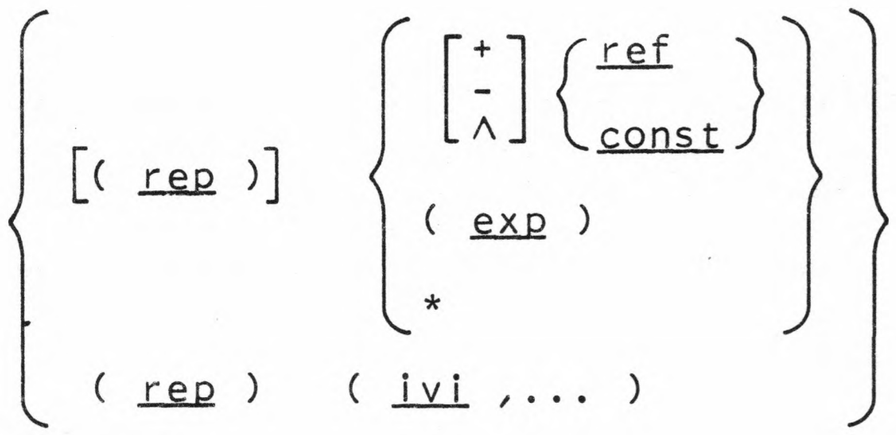
The INITIAL attribute is used with the name of a scalar or array variable in order to set the value of that variable when the variable is allocated.

Initialization Syntax

The syntax of the INITIAL attribute is given by the following diagram:



where ivi (initial value item) is defined as



and where

- rep (replicator) is any expression whose value can be converted to an integer.
- ref is any variable reference or function reference.
- const is a literal constant or is constant complex expression (a real, a sign, and an imaginary).
- exp is any expression.
- ivi,... is a sequence of one or more initial value items separated by commas.

According to the syntax, there are two forms for an initial value item. The first form is the one that provides a single initial value or, if a replicator is used, a sequence of identical values. The syntax makes a special provision for references and constants in order to exempt them from being enclosed in parentheses. Consider the following attribute:

```
INITIAL(-2.8-15I,(N-1)0)
```

The first item specifies a single initial value; the second item specifies a sequence of zero values whose length depends on the value of N when storage is allocated for the variable to which the attribute applies. Consider next the attribute:

```
INITIAL(X,-V(I+2,J),(4)F(A-SIN(THETA,Z)))
```

This attribute could be used to initialize a six-element array (although such variety is unlikely for the initialization of a single array). The first item is a variable name, the second is a subscripted variable reference, and the third is a function reference that is evaluated and has four copies of its value entered in the list.

The use of an asterisk, '*', as an initial value causes the corresponding value to be left undefined; that is, it skips initialization. Consider the following:

```
INITIAL((5)(1,*))
```

If this attribute is applied to an array with ten elements, it will initialize the odd-numbered elements to 1 and leave the others undefined.

The second form of initial value item allows the use of a parenthesized list of items as a sublist of a larger list. In this form, the replicator is required because the only reason for forming a sublist is to replicate it as a whole. Consider, for example, the attribute:

```
INITIAL((3)(-1,(2)0))
```

After application of the first replicator, this attribute is equivalent to:

```
INITIAL(-1,(2)0,-1,(2)0,-1,(2)0)
```

After expansion of the remaining replicators, this attribute is equivalent to:

```
INITIAL(-1,0,0,-1,0,0,-1,0,0)
```

When the replicator is a variable expression, it cannot be expanded until its values are actually required; for example:

```
INITIAL((N)(-1,(M+3)0))
```

depends on the values of N and M.

The Use of the INITIAL Attribute

An INITIAL attribute cannot be used with a structure name. It follows that the attribute can only be used with a variable name that designates a scalar or an array of scalars. When the attribute is used with a scalar, it must supply exactly one value; when it is used with an array, it must provide one value for each element of the array. Each value must be suitable for assignment to the variable it initializes.

As an example of several uses of the INITIAL attribute, consider the following declaration:

```
DCL 01 A,
       02 B(M,N-2) FLOAT INIT((M)(1,(N-3)*),
       02 C CHAR(3) INIT("XXX"),
       02 D,
          03 E FLOAT INIT(0),
          03 F POINTER INIT(NULL()),
       02 G FLOAT;
```

In this structure, the array A.B has its first column initialized to 1 and its remaining columns left undefined, the scalar variables C, E, and F are initialized to appropriate scalar values; and the scalar variable G is not initialized.

An INITIAL attribute is processed only as part of the storage allocation operation. Since a variable name of storage class PARAMETER or DEFINED is never subjected to the storage allocation (but rather is associated with existing storage) it is incorrect to use an INITIAL attribute with such names. A variable name of storage class BASED can be used either to allocate storage, in which case its INITIAL attribute is processed, or to be associated with existing storage, in which case the INITIAL attribute is ignored.

In the definition of string literal constants, given later in the section on "Constants", a replicator is allowed for a string constant. For example:

```
(3)"AB"
```

is equivalent to "ABABAB". The PL/I processor will interpret an initial value replicator as a string replicator if it occurs in an appropriate context. For example, the declaration

```
DCL S(3) CHAR(2) INIT((3)"AB");
```

is invalid because the initial attribute is expanded to give the wrong result:

```
DCL S(3) CHAR(2) INIT("ABABAB");
```

To obtain the desired result, the programmer must write the string replicator (1) and then precede that with an initial value replicator:

```
DCL S(3) INIT((3)(1)"AB");
```

This problem is a flaw in the design of PL/I; it arises only in the initialization of string variables, and the incorrect usage illustrated above is detected by the compiler.

THE CAPACITY OF STORAGE

A GCOS program is restricted to about a fifth of a million words (of 36 bits or 4 characters each). The total space occupied by the following components of a PL/I program must therefore be less than this limit.

- Instructions and constants generated by the PL/I processor.
- Static storage assigned by the PL/I processor for internal static variables and by the loader for external static variables.

- Library routines called by generated instructions and provided by the loader.
- The automatic data stack for automatic variables.
- System storage for explicitly allocated variables.

The size of the space required for the first two items is announced by the PL/I processor for each external procedure called in the program. The size of the library routines is announced by the loader; it varies from a minimum of ten thousand words to thirty or forty or more thousand words depending on the particular arithmetic and file routines called. The sizes of the last two items cannot be determined prior to execution since they depend on the relative time of block activations and ALLOCATE and FREE statement executions.

An estimate of the maximum size required for stack and system storage at any one time during execution can be made using the stack size announced by the PL/I processor for each external procedure and the sizes of variables allocated storage by means of an ALLOCATE statement. The user specifies the number of words required for all of the five items for the program on a $ LIMITS control statement (refer to the section on "Execution of a PL/I Program" in the PL/I User's Guide). The program will not be loaded unless the limit specification is large enough for the first three items. Once loaded, the program seeks more space, beyond the specified limit, if necessary, to provide stack or system storage. As long as its requests for more space are honored by the operating system, as they ordinarily are, the estimation of stack and system storage sizes is not crucial.

Because stack and system storage allocations are made from opposite ends of the space remaining after the static storage requirements are satisfied, however, it is possible for the requesting of more storage to be insufficient. A system storage allocation can be made that uses space adjacent to the stack space, and until that allocation is freed, the stack space cannot be increased. To avoid this blockage, system storage allocations are prevented from using space close to the current stack bottom. When this allowance is not sufficient, however, the program will report a storage condition and be aborted. The user should then increase the limits specification and rerun the program.

For other discussions of the memory layout and size requirements of a PL/I program, refer to the section on "Debugging PL/I Programs" and the appendix on "Memory Requirements" in the PL/I User's Guide.

CONDITIONS FOR STORAGE MANAGEMENT

As a result of the allocation operation, certain conditions can occur. The purpose of such a condition is to report that there is no storage available for use by the allocated variable. A program can include ON statements to respond to such a condition by performing a remedial action (such as freeing some storage). If the program does not have ON statements for this purpose, the PL/I processor outputs an error message and terminates program execution. Details are given later, in the section on "Condition Handling".

The STORAGE Condition

When the STORAGE condition occurs, one of the following cases applies:

- The stack segment is nearly filled. The stack segment is used for activation regions. It can be exhausted either by allocation of a very large automatic variable or by a runaway recursive execution of a procedure. The simplest remedial action is to execute a GOTO to a statement outside the procedure being executed (or perhaps outside a nest of procedures and blocks); this action causes the destruction of the activation regions associated with the procedure and thus frees storage in the stack segment.

- The system storage segment is nearly filled. The system storage segment is used for controlled variables and explicitly allocated based variables. The remedial action is to free some of these variables.

In most programs, no provision is made for remedial action for these conditions; that is, their occurrence is usually considered to be a programming error.

The STORAGE condition takes care of the classes of storage that can be allocated dynamically: automatic, controlled, and based. A condition is provided for these storage classes because they can cause storage overflow in one execution of a program but not in another. Storage can be allocated for one other class of storage: static. No condition is provided for that class because if overflow of static storage occurs for any execution of a program it will occur for all executions of the program, and therefore the program is invalid; furthermore, there is no way to free static storage and thus no remedial action for overflow.

The AREA Condition

The AREA condition occurs when an attempt is made to allocate storage in an area variable that cannot supply the storage. Two cases apply:

- An ALLOCATE statement attempts to allocate a based variable for which there is no room in the area variable. A valid remedial action is to free some or all of the storage in the area variable (usually after copying the values into other storage or outputting them to a file). After this action, it is valid to resume program execution at the point of interruption.

- An assignment statement attempts to assign an area value to an area variable that is too small. The remedial action is to transfer to some other place in the program; it is not valid to return to the point of interruption.

The AREA condition can play an important role in large scale programs for the manipulation of linked data structures.

SECTION VIII

EXPRESSIONS

An expression is used in a program wherever a value is required. Some expressions are very simple; for example, the variable reference X and the constant literal 25 are both expressions. Some expressions combine several operators to calculate a value; for example, '2*ALPHA*(X-Y)' is an expression that contains three operators. Some expressions invoke procedures; for example, F(X) yields a value that is obtained by invoking a procedure at entry point F with the argument X.

The rules for the interpretation of expressions are complicated. One source of complexity is the requirement that each expression have an associated storage type; there are many storage types, and the rules for their use vary from one kind of expression to another. Further complexity arises from the fact that expressions can have aggregate values. Yet another complication arises in the interpretation of references to BASED and DEFINED variables.

A programmer who is learning PL/I can eliminate some of the complexity by ignoring features that he does not need. He may be able to avoid fixed-point arithmetic except for integer counters and subscripts; that simplifies the use of built-in functions and operators. He may be able to avoid aggregate expressions except, perhaps, for the assignment of one variable to another; that simplifies the rules for determining the storage type of expressions. He may be able to avoid BASED and DEFINED variables; that eliminates the difficult rules for equivalencing variable names. By thus selecting a subset of the expressions, the programmer can skip over some of the more complicated rules.

In this section, the six kinds of expressions are listed and the features they have in common are described. Then each kind of expression is described separately and in detail.

GENERAL REMARKS ON EXPRESSIONS

An expression is one of the following six constructs:

variable reference
constant literal
constant reference
programmed function reference
built-in function reference
operator expression

Before these kinds of expressions can be discussed individually, some general features of expressions must be discussed. The following paragraphs consider the nesting and parenthesization of expressions, the determination of storage types, the use of aggregate expressions, and the rules for ordering and optimizing the evaluation of expressions. In the course of these general remarks, several examples of expressions are discussed in detail.

Nested Expressions

In some cases, an expression can contain other expressions. Specifically, an operator expression has expressions as its operands; a function reference has expressions as its arguments; a subscripted variable reference has expressions as its subscripts; and a locator-qualified variable reference has an expression as its locator-qualifier. Such a use of one expression within another is called nesting. There is no restriction on nesting, and an expression can contain an expression that contains another expression that contains another expression and so on, to any reasonable depth of nesting.

An example of nested expressions appears in the following program:

```
P:   PROC;
        DCL (X,U) FLOAT;
        DCL (I,J) FIXED;
        DCL A(10,10) FLOAT;
        DCL SIN BUILTIN;
        ...
        X = SIN(U)*A(I+2,J);
        ...
        END;
```

The right-hand-side of the assignment statement is constructed of eight expressions, as follows:

- The entire expression is an operator expression, and it has SIN(U) and A(I+2,J) as its operands.
- The expression SIN(U) is a built-in function, and it has U as its argument.
- The expression A(I+2,J) is a subscripted variable reference, and it has I+2 and J as its subscripts.
- The expression I+2 is an operator expression, and it has I and 2 as its operands.
- The expressions U and I and J are simple variable references.
- The expression 2 is a constant literal.

Observe that the nesting in this example covers four levels: the product contains a subscripted variable reference that contains a sum that contains a simple variable reference.

Parenthesized Expressions

Any expression can be enclosed in parentheses. There are two situations in which the use of parentheses is appropriate, as follows:

- A parenthesized expression is used to modify the order in which operators are evaluated. For example, consider the expression:

  ```
  2*(A+B)
  ```

  The operator priority rules of PL/I, given later in this section, specify that the '*' operation is performed before the '+' operation. However, the parentheses in this expression cause the '+' operation to be performed first.

- A parenthesized expression is sometimes used as an argument in a function reference or a CALL statement; specifically, it is used when it is necessary to force an argument that would otherwise be interpreted as by-reference to be interpreted as by-value. This is a specialized use of a parenthesized expression; it is discussed in the section on "Procedure Invocation".

The storage type and value of a parenthesized expression are the same as those of the enclosed expression.

Storage Types of Expressions

The interpretation of an expression yields a value and storage type. The value of an expression is determined each time the expression is evaluated during program execution. In contrast, the storage type is determined once, by the compiler, before the program is executed.

Storage types were defined earlier, in the section on "Value Storage". The storage type of an expression determines the way the value of an expression is represented. Because the storage type is determined in advance, the compiler can provide exactly the required storage for the value and can generate instructions that are appropriate for the value.

As a basis for the discussion of the storage types of expressions, consider the following program:

```
P:   PROC;
DCL I FIXED;
DCL (X,THETA) FLOAT;
DCL B(20) FLOAT;
DCL COS BUILTIN;
...
X = B(I-3)*COS(THETA);
...
END;
```

The right-hand-side of the assignment statement is made up of seven expressions. The storage type of each expression is obtained as follows:

- The storage type of the variable reference I is REAL FIXED BIN(17,0). It is obtained from the first DECLARE statement according to the rules for a simple variable reference given later in this section.

- The storage type of the constant literal 3 is REAL FIXED DEC(1,0). It is obtained from inspection of the constant literal itself according to rules given later in this section. Those rules specify that the constant literal is REAL because it does not have an I at the end, FIXED because it does not have an exponent, DECIMAL because it does not have a B at the end, and PRECISION(1,0) because it has one digit and that digit is not a fractional digit.

- The storage type of the expression 'I-3' is REAL FIXED BIN(18,0). It is obtained from the rules for the '-' operator, which are given later, in the section on "Operations". According to those rules, the constant literal is first converted to a REAL FIXED BIN(4,0) value; this can be done during compilation. The number-of-digits of the result is 18 in order to allow for the possibility of a carry from the subtraction operation.

- The storage type of the subscripted variable reference B(I-3) is REAL FLOAT BIN(27). It is obtained from the third DECLARE statement according to the rules given later in this section. The subscript in the reference cancels out the extent in the declaration, so that the storage type of the reference is <u>scalar</u>.

- The storage type of the variable reference THETA is REAL FLOAT BIN(27). It is obtained from the second DECLARE statement according to the rules for a simple variable reference.

- The storage type of the built-in function reference COS(THETA) is REAL FLOAT BIN(27). It is obtained from the rules for the COS built-in function, which are given later, in the section on "Operations". In this simple case, the storage type of the result is the same as the storage type of the argument.

- The storage type of the entire expression is REAL FLOAT BIN(27). It is obtained from the rules for the '*' operation, which are given later, in the section on "Operations". Observe that the storage type of the result is, in this case, the same as the storage type of the operands.

This example is typical in several respects. It shows that even a simple use of fixed-point arithmetic has some tricky points. It shows the simplicity of floating-point calculations, which often carry the single storage type REAL FLOAT BIN(27) straight through the calculation. And it shows how an aggregate is referenced to obtain a scalar value.

Aggregate Expressions

The main part of this section is devoted to defining the six kinds of PL/I expressions. In the definition of each kind of expression, aggregate values are mentioned. That aspect of the definition of expressions is summarized here:

- A variable reference or a programmed function reference can have any storage type and therefore can have any aggregate type. This generality permits aggregate values to be handled as single entities when they are subjected to input/output, are copied from one variable to another, or are operated upon by a procedure.

- A constant literal or a constant reference cannot (except for LABEL constant references) have an aggregate type because PL/I does not include a way of writing an aggregate constant. This omission reflects a decision of the designers of PL/I rather than a fundamental limitation of programming languages. It is quite easy to program around this deficiency.

- Most built-in function references and operator expressions can have aggregate types; in such a case, the aggregate type of the result is derived from the aggregate types of the arguments or operands. Operations on aggregates are performed on their respective components; that is, an operation is applied to the ith component of each argument to produce the ith component of the result.

The aggregate variable references and programmed function references can be especially useful in writing a clear and efficient program. In contrast, the aggregate built-in function references and operator expressions are rarely useful.

Ordering and Optimizing the Evaluation of Expressions

The definition of PL/I deliberately leaves undefined certain aspects of expression evaluation. In fact, the only general rule for expression evaluation is:

- When the value of an expression is required, it is evaluated at some time after the last computation that could change the value of the expression and at some time before the computation that requires its value.

The vagueness of this rule permits the compiler to determine the details of the evaluation of an expression and thus produce optimized code for the program. For example:

- When the same expression appears in several places, the compiler can evaluate it just once if the compiler can determine that the value of the expression does not change between the given appearances.

- The subscripts of a variable reference can be evaluated in any order. The same freedom applies to the arguments of a function reference or the operands of an operator expression.

- An argument or operand can be ignored if it does not affect the result of an operation. For example, the second operand of an "and" operation can be ignored when the value of the first operand is "false".

Thus there can be more than one way to evaluate a given expression. However, the variations that are permitted are chosen so that, in most cases, they have no effect on the results.

Some consideration must be given to the cases in which the undefined aspects of expression evaluation do affect the results of program execution. These cases arise because of side effects. A side effect is a change in the value of a variable or in the environment of the program that is caused by the evaluation of an expression. There are two kinds of side effects:

- The evaluation of a programmed function reference can invoke a procedure that produces a side effect.
- The evaluation of an expression can cause a condition to occur that signals an ON unit that produces a side effect.

It is easy to recognize the possibility of a side effect produced by a programmed function reference. It is not so easy to fully appreciate the possibilities of side effects from the occurrence of conditions. Such conditions as SIZE, FIXEDOVERFLOW, UNDERFLOW, and OVERFLOW can occur almost anywhere in the evaluation of an expression. Thus the side effects caused by an ON unit are especially significant.

As an example of a program whose results are partially undefined, consider the following:

```
P:   PROC;
        DCL (X,Y) FLOAT;
        DCL (SYSIN,SYSPRINT) FILE;
        GET LIST(X,Y);
        PUT LIST(F(XF(Y));
F:      PROC(A) RETURNS(FLOAT);
        DCL A FLOAT;
        PUT LIST(A);
        RETURN(A**2);
        END;
     END;
```

The first PUT statement in this program must evaluate the expression:

```
F(X)+F(Y)
```

Each operand in this expression is a programmed function reference that has a side effect. The side effect is the listing of the value of the argument of the programmed function reference; that action changes the environment of the program. Because the order in which the operands of '+' are evaluated is undefined, the order in which the argument values are listed is undefined. Therefore, the result of executing the program is partially undefined. Nevertheless, the program is valid and it is also correct unless the programmer cares about the order in which X and Y are listed.

The results of executing a program are not necessarily undefined just because some part of the execution of the program is undefined. For example, suppose the statement:

```
PUT LIST(A);
```

is removed from the program given in the previous paragraph. After this change, the evaluations of the programmed function references have no side effects. The execution of the program is still undefined because the order in which the operands of '+' are evaluated is still undefined. However, the result of executing the program is fully defined: the program lists a single value that is the sum of the squares of X and Y.

VARIABLE REFERENCES

The interpretation of a variable reference begins with the determination of two items of information: the location of a variable in storage and the storage type of the variable. For some variable references, this information is easy to obtain; for others, it requires a careful interpretation of the reference. Once this information has been obtained, the remainder of the interpretation of the variable reference depends on whether the variable reference is used as an expression or as the target of an assignment. If the variable reference is used as an expression, then the value of the designated variable is retrieved and becomes the value of the reference. If the variable reference is used as the target of an assignment, then the assigned value is converted to the designated type and is placed in the designated variable.

Preliminary Remarks on Variable References

There are four kinds of variable references, as follows:

simple
subscripted
structure-qualified
locator-qualified

This list is given in order of increasing complexity and decreasing frequency of use. Thus, for example, a locator-qualified variable reference has a complicated interpretation but is used only in special programming applications.

Each of the four kinds of variable reference is described in detail in this section. Each description depends on the preceding descriptions; for example, the description of subscripted references makes use of rules that are given in the description of simple references.

THE MAJOR NAME IN A VARIABLE REFERENCE

A variable reference begins with a major name unless it is shortened or locator-qualified. The major name designates a variable in storage that is not part of a larger variable, and the remainder of the reference indicates the portion of the variable that must be retrieved. Consider, first, a simple variable reference:

```
SPEED3
```

In this case, the major name is the entire variable reference, so the value of the reference is the value of the entire variable. Consider, next, a subscripted variable reference:

```
A(3)
```

In this case, the major name A designates an array variable and '(3)' designates an element of that array variable. Consider, finally, a structure-qualified variable reference:

```
ALPHA.Q(I,J-3)
```

In this case, the major name ALPHA designates a structure variable and .Q(I,J-3) designates an element of a two-dimensional array that is a member of the structure variable.

When a variable reference is shortened, the major name may be missing. For example, in certain contexts the structure-qualified variable reference given in the preceding paragraph can be shortened to:

```
Q(I,J-3)
```

In this case, Q is not the major name. Instead, the reference must be expanded to its original, complete form before it can be interpreted; and then ALPHA is the major name, as before.

The interpretation of a locator-qualified variable reference is quite different from the other three kinds of variable reference. In a locator-qualified reference, the place of the major name is taken by a locator-qualifier. The locator-qualifier, like the major name, designates a variable in storage; however, it can be any reference that yields a locative value. Thus the designation is computed at the time the reference is interpreted instead of being given, as a name, once and for all when the program is written. This general facility is useful for list-processing applications of PL/I.

Simple Variable References

A simple variable reference designates a major variable. The variable can be either a scalar or an aggregate.

THE FORM OF SIMPLE VARIABLE REFERENCES

A simple variable reference has the following form:

id

where *id* must be an identifier that is declared as the name of a major variable and that is not declared BASED.

As an example of a simple variable reference, consider:

```
ALPHA
```

This name must have a declaration of the form:

```
DCL ALPHA ... ;
```

or the form

```
DCL 01 ALPHA ... ,
       02 ...,
       ... ;
```

where the '...' symbols indicate portions of the declarations that are not significant for this discussion.

THE INTERPRETATION OF SIMPLE VARIABLE REFERENCES

A simple variable reference that is used for retrieval is interpreted as follows:

1. Name Resolution. Resolve the variable name. (The resolution of names is described earlier, in the section on "Declaration of Identifiers".) The result is the declaration of the variable name.

2. Storage Type Determination. Obtain the storage type of the variable name from its declaration. The result is the storage type for the given reference.

3. Variable Location. Locate the variable designated by the variable name.

4. Value Retrieval. Retrieve the value of the designated variable.

Most of the interpretation is performed by the compiler; only the last step, value retrieval, is performed during program execution.

The location of the variable, mentioned in Step 3 of the interpretation, depends not only on the name of the variable but also on its storage class and scope attributes, as follows:

- If the name is STATIC EXTERNAL, then the variable is in the external region.

- If the name is STATIC INTERNAL, then the variable is in the permanent internal region associated with the block in which the name is declared.

- If the name is CONTROLLED EXTERNAL, then the variable is in the external region. If more than one variable has been allocated for the given name, then the most recently allocated variable is used.

- If the name is CONTROLLED INTERNAL, then the variable is in the permanent internal region associated with the block in which the name is declared. If more than one variable has been allocated for the given name, then the most recently allocated variable is used.

- If the name is AUTOMATIC INTERNAL, then the variable is in the activation internal region that corresponds to the block in which the name is declared. If the program is recursive, then there can be more than one activation internal region for a given block; the selection of one of these regions is made according to rules given in the section on "Procedure Invocation".

- If the name is declared PARAMETER INTERNAL, then the variable is reached by first locating a POINTER temporary that is designated by the name and that is in the activation internal region that corresponds to the block in which the name is declared. Then the pointer is followed to the desired variable. Details are given in the section on "Procedure Invocation".

- If the name is declared DEFINED INTERNAL, then the variable is located by rules that are given in the section on "Storage Management".

Once the appropriate storage region has been located, the variable is uniquely designated by the given name.

EXAMPLES OF SIMPLE VARIABLE REFERENCES

Examples of simple variable references are discussed here; they occur in the following program:

```
P:   PROC;
        DCL X DEC(6,3);
        DCL 01 Y,
               02 FLAGS BIT(6),
               02 SIDE(2) DEC(5);
        DCL (SYSIN,SYSPRINT) FILE;
        ... (assignments to variables occur here)
        PUT LIST(X,Y,FLAGS,SIDE);
        ...
        END;
```

When execution of the program begins, the variables X and Y are allocated; then, when execution of the program is underway, values are assigned to the variables. Suppose that before the example output statement is executed, storage includes the region:

activation internal region, example program

```
                 S 9 9 9   9 9 9
00113    X      /-/0/0/2/./3/6/3/

                           1 1 1 1 1 1
02022    Y   .FLAGS     "/1/1/1/0/0/1/"B

                         S 9 9 9 9 9
             .SIDE(1)   /+/0/0/0/1/2/

                         S 9 9 9 9 9
                 (2)    /+/0/0/0/0/8/
```

Four variable references appear in the output statement in the example program:

- X is a true simple reference (that is, it is not a shortened form of some other reference) and it yields:

  ```
  storage type:     REAL FIXED DEC(6,3)
  value:            -2.363
  ```

- Y is also a true simple reference, and it yields:

  ```
  storage type:     01, 02 BIT(6) NONVARYING,
                        02 DIM(2) REAL FIXED DEC(5,0)
  value:            "111001"B, 12, 8
  ```

- FLAGS is a shortened form of the structure-qualified reference Y.FLAGS and is interpreted accordingly (as described under "Structure-Qualified Variable References" later in this section).

- SIDE is a shortened form of Y.SIDE(*)', which also is a structure-qualified reference.

Subscripted Variable References

A subscripted variable reference designates a portion of a major array variable. The designated portion can be either a single element of the array, in which case it is a scalar, or a cross section of the array, in which case it is, itself, an array. The designated portion of the major variable is specified by one or more subscripts.

THE FORM OF SUBSCRIPTED VARIABLE REFERENCES

A subscripted variable reference has the following form:

mn(sublist)

where mn is the major name and sublist is the subscript list. The major name must be an identifier that is declared as the name of a major variable and that is not declared BASED. The subscript list is a sequence of subscripts separated by commas, and each subscript is either an expression or an '*'. A subscript expression must yield a value that can be converted to an integer. A subscripted variable reference that has one or more '*' subscripts designates a cross-section of an array.

As an example of a subscripted variable reference, consider:

```
PHI(I+CEIL(.362*SQRT(X-1)),*,J-2)
```

In this example the major name is PHI and there are three subscripts. The first subscript is chosen to show that there is no special restriction on a subscript expression. The major name must have a declaration of the form:

DCL PHI(dim , dim , dim) ... ;

where each dim represents a dimension and '...' represents a sequence of attributes.

THE INTERPRETATION OF SUBSCRIPTED VARIABLE REFERENCES

A subscripted variable reference that is used for retrieval of a value is interpreted as follows:

1. Name Resolution. Resolve the major name in the given reference. The result is the declaration of the major name.

2. Storage Type Determination. Make a copy of the storage type of the major name and delete from the DIMENSION attribute of the name each dimension that corresponds to a subscript that is an expression. Do not delete a dimension that corresponds to an '*' subscript. If all dimensions are deleted, then delete the DIMENSION attribute. The result is the storage type of the reference.

3. Subscript Evaluation. Evaluate each subscript expression in the reference and, if necessary, convert its value to an integer. If a subscript value is outside the range of subscripts for which the array variable is allocated, the SUBSCRIPTRANGE condition occurs. The result of subscript evaluation is the fully-bound reference.

4. Variable Location. Locate the variable designated by the major name; do this just as for a simple variable reference. The designated variable is the sequence of storage units that are selected by the fully-bound reference. A storage unit is selected if it matches the beginning of the designator or the entire designator of the storage unit. The match must be exact except that an '*' in the reference matches any integer subscript in a storage unit designator.

5. Value Retrieval. Retrieve the value of the designated variable.

Most of the interpretation is performed by the compiler; only subscript evaluation and value retrieval are performed during program execution.

The interpretation just given for a subscripted variable reference is complete, but it requires the following remarks to clear up difficult points:

- The determination of the storage type by Step 2 and the retrieval of the value by Step 4 are consistent with one another. That is, the interpretation of the reference guarantees that the value retrieved is always appropriate for the storage type of the reference.

- The SUBSCRIPTRANGE condition mentioned in Step 3 is part of the mechanism provided by the language to detect errors or exceptions that occur during program execution. The programmer can supply statements to handle such a condition; more often, he allows the interpreter to report it as an error and abort the program. There is a cost associated with checking the value of a subscript, and the language allows the programmer to conveniently enable this check during program debugging and then disable it when the program enters production. Details are given later, in the section on "Condition Handling".

- In general, a subscripted variable reference selects a subset of the elements of the array variable that is designated by the major name. When a subscript is an expression (and is evaluated to produce an integer), it participates in making the subset smaller. On the other hand, when a subscript is an '*', it makes no contribution to the selection and allows any subscript value to match its subscript position.

Two special cases of the subscripted variable reference are of particular interest:

- The most common use of a subscripted variable reference is that in which the major name is declared to be an array of scalars and the subscript list contains no '*'. In this case, the storage type of the reference is scalar and the fully-bound reference designates a single storage unit.

- When every subscript is an '*', the reference designates the entire variable designated by the major name. Specifically, it follows from Step 2 that no dimension is deleted from the storage type and it follows from Step 4 that the fully-bound reference matches the designator of every storage unit of the array variable.

EXAMPLES OF SUBSCRIPTED VARIABLE REFERENCES

Examples of subscripted variable references are discussed here; they occur in the following program:

```
P:   PROC;
        DCL A(3,2) DEC(4);
        DCL 01 PART(0:1),
               02 NAME CHAR(6),
               02 CODE DEC(5);
        DCL SYSPRINT FILE;
        DCL (I,J,M) FIXED;
        DCL X FLOAT;
        ... (assignments to variables occur here)
        PUT LIST(A(I+2,J-X**2),A(I+2,*),A(*,J-X**2),A(*,*));
        PUT LIST(PART(M),PART(*));
        ...
        END;
```

Suppose that before the example output statements are executed, storage includes the region:

activation internal region, example program

```
00242    A    (1,1)   S 9 9 9 9
                      /+/0/2/0/1/

00244         (1,2)   S 9 9 9 9
                      /+/0/2/0/2/

00246         (2,1)   S 9 9 9 9
                      /+/0/2/0/3/

00250         (2,2)   S 9 9 9 9
                      /+/0/2/0/4/

00252         (3,1)   S 9 9 9 9
                      /+/0/2/0/5/

00254         (3,2)   S 9 9 9 9
                      /+/0/2/0/6/

03544   PART    (0).NAME   X X X X X X
                           "/T/X/8/Q/2/4/"

03546               .CODE  S 9 9 9 9 9
                           /+/0/0/5/9/3/

03550          (1).NAME   X X X X X X
                           "/W/6/R/F/E/D/"

03552               .CODE  S 9 9 9 9 9
                           /+/8/0/0/0/7/
```

Subscripted references to the same variable differ in an important way when one reference has an expression for a subscript and the other has an '*' for the same subscript. According to this view, four different references can be written for the variable A, as follows:

- A(I+2,J-X**2) has expressions as subscripts. Suppose that the reference is interpreted when

  I = 0, J = 5, and X = 2

  Then the fully-bound reference is A(2,1). The result of the interpretation of the reference is:

  ```
  storage type:     REAL FIXED DEC(4,0)
  value:            203
  ```

- A(I+2,*) is a one-dimensional cross-section of the two dimensional array variable. Suppose the value of I is as before; then the fully-bound reference is A(2,*). The result of the interpretation of the reference is:

  ```
  storage type:     DIM(2) REAL FIXED DEC(4,0)
  value:            203, 204
  ```

- A(*,J-X**2) is also a one-dimensional cross-section of the two dimensional array variable. Suppose the values of J and X are as before; then the fully-bound reference is A(*,1). The result of the interpretation of the reference is:

  ```
  storage type:     DIM(3) REAL FIXED DEC(4,0)
  value:            201, 203, 205
  ```

- A(*,*) is a two-dimensional cross-section of the two dimensional array variable; that is, it designates the entire array variable. The result of the interpretation of the reference is:

  ```
  storage type:     DIM(3,2) REAL FIXED DEC(4,0)
  value:            201, 202, 203, 204, 205, 206
  ```

The variable PART is an array of structures. There are two different references for the variable, as follows:

- PART(M) has an expression as its subscript. Suppose the reference is evaluated when

  M = 0

  Then the fully-bound reference is PART(0). The result of the interpretation of the reference is:

  ```
  storage type:     01, 02 CHAR(6) NONVARYING,
                       02 REAL FIXED DEC(5,0)

  value:            "TX8Q24", 593
  ```

- PART(*) designates the entire variable. The result of interpretation of the reference is:

  ```
  storage type:     01 DIM(1), 02 CHAR(6) NONVARYING,
                       02  REAL FIXED DEC(5,0)

  value:            "TX8Q24", 593, "W6RFED", 80007
  ```

Structure-Qualified Variable References

A structure-qualified variable reference designates a portion of a variable that is a structure or an array of structures. The designated portion of the variable can either be a scalar or another, smaller, aggregate; it is selected by level names and, in some cases, subscripts.

THE FORM OF STRUCTURE-QUALIFIED VARIABLE REFERENCES

A structure-qualified variable reference has one of the following forms:

lref1 . lref2

lref1 . lref2 . lref3

... (and so on)

where lref1, lref2, lref3, and so on, are level references. Each level reference is a level name optionally followed by a parenthesized subscript list. The subscript list is a sequence of subscripts separated by commas, and each subscript is either an expression or an '*'. A subscript expression must yield a value that can be converted to an integer.

The rightmost level reference is the member reference and the other level references are containing references. The leftmost containing reference must begin with a major name that designates an aggregate variable. The structure-qualified reference as a whole must be consistent with the declaration of the major name; that is, the first-level reference must designate a second-level component of the major variable, the second-level reference must designate a third-level component, and so on.

An example of a structure-qualified reference is:

```
BASE.FIRST
```

In this example, there are two level references, each in the form of a simple reference. The name BASE is a containing reference and is the major name for the structure-qualified reference as a whole. The name FIRST is the member reference. The identifiers must be declared by a statement of the form:

```
DCL 01 BASE ... ,
       ...
       02 FIRST ...,
       ... ;
```

where the '...' symbols indicate portions of the statement that are not of interest here.

A second and more complicated example of a structure-qualified reference is:

```
X(J,*).Y3_TEST(2*M-3/I).PAR4
```

In this example, there are three level references: the first two, the containing references, are in the form of subscripted references and the last one, the member reference, is in the form of a simple reference. The name X is the major name for the reference and it must be declared as a level-one, two-dimensional array of structures. The name Y3_TEST must be declared as a one-dimensional array of structures that is a member of X. The identifier PAR4 must be declared as a member of Y3_TEST. In other words, the following DECLARE statement must apply:

```
DCL 01 X( dim , dim ) ... ,
       ... ,
       02 Y3_TEST( dim ) ...,
          ... ,
          03 PAR4 ... ,
          ... ;
```

Observe that although the reference places constraints on the aggregate type of the containing references, it places no constraint on the member reference PAR4; so PAR4 could be a scalar, array, or structure.

THE INTERPRETATION OF STRUCTURE-QUALIFIED VARIABLE REFERENCES

A structure-qualified reference that is used for retrieval of a value is interpreted as follows:

1. Name Resolution. Resolve the major name in the given reference. The result is the declaration of the major name.

2. Storage Type Determination. Perform the following steps:

   a. Make a copy of the normalized storage type for the major name of the given reference.

   b. Edit the dimensions in the storage type as follows:

      (1) Delete each dimension that is associated with a subscript expression in the given reference.

      (2) Keep each dimension that is associated with an '*' subscript in a containing reference of the given reference, but move it so that it occurs in the storage-type component that corresponds to the member reference.

      (3) Keep all other dimensions.

   c. Edit the remainder of the storage type as follows:

      (1) Keep the component of the storage type that corresponds to the member reference.

      (2) If the member reference designates a structure, keep each component of the storage type that corresponds to a component of that structure.

      (3) Delete the remaining components of the storage type.

The result is the storage type of the reference. It is understood that, as the storage type is derived by the steps above, the necessary refinements are supplied to keep the storage type in a valid, normalized form. For example, when the last dimension in a DIMENSION attribute is deleted, the DIMENSION attribute itself is deleted.

3. Subscript Evaluation. Subscripts are evaluated just as already described for a subscripted variable reference. The result is the fully-bound reference.

4. Variable Location. Locate the variable designated by the given major name; do this just as for a simple variable reference. The designated variable is the sequence of storage units that are selected by the fully-bound reference. A storage unit is selected by the fully-bound reference. A storage unit is selected if it matches the beginning of the designator or the entire designator of the storage unit. The match must be exact except that an '*' in the reference matches any integer subscript in a storage unit designator.

5. Value Retrieval. Retrieve the value of the designated variable.

Most of the interpretation is performed by the compiler; only subscript evaluation and value retrieval are performed during program execution.

This interpretation, especially the determination of the storage type in Step 2, is complicated. Rather than discuss it in the abstract, a detailed discussion of an example reference is given in what follows.

EXAMPLES OF STRUCTURE-QUALIFIED VARIABLE REFERENCES

Examples of structure-qualified references are discussed here; they occur in the following program:

```
P:   PROC;
        DCL 01 Q(2) STATIC EXTERNAL,
               02 R1,
                  03 S1(4) FLOAT,
                  03 S2 CHAR(4),
               02 R2(0:1,3) DEC(10);
        DCL SYSPRINT FILE;
        ... (assignments to variables occur here)
        PUT LIST(Q(*).R1.S1(I+3));
        ...
        END;
```

A diagram of the storage for Q would be of inconvenient size. Instead, a complete list of the designators for the storage units that make up the variable is given:

```
Q(1).R1.S1(1),  Q(1).R1.S1(2),  Q(1).R1.S1(3),  Q(1).R1.S1(4),
Q(1).R1.S2,
Q(1).R2(0,1),  Q(1).R2(0,2),  Q(1).R2(0,3),
Q(1).R2(1,1),  Q(1).R2(1,2),  Q(1).R2(1,3),
Q(2).R1.S1(1),  Q(2).R1.S1(2),  Q(2).R1.S1(3),  Q(2).R1.S1(4),
Q(2).R1.S2,
Q(2).R2(0,1),  Q(1).R2(0,2),  Q(2).R2(0,3),
Q(2).R2(1,1),  Q(2).R2(1,2),  Q(2).R2(1,3)
```

The interpretation of a structure-qualified reference is now given in detail. Suppose the reference

```
Q(*).R1.S1(I+3)
```

is interpreted when I = -1. The steps in the interpretation are:

1. (Name Resolution.) The DECLARE statement in the example program is associated with Q.

2. (Storage Type Determination.) The following steps are performed:

   a. A copy of the normalized storage type is made, giving:

```
01 DIM(2),
   02,
      03 DIM(4) FLOAT,
      03 CHAR(4),
   02 DIM(2,3) DEC(10)
```

   Observe that the normalized form of the dimension '0:1' (declared for R2) is 2.

   b. The dimension is edited. The dimension associated with 'I+3' is omitted, giving:

```
01 DIM(2),
   02,
      03 FLOAT,
      03 CHAR(4),
   02 DIM(2,3) DEC(10)
```

   Then the dimension associated with the '*' subscript in the first containing reference is moved to the storage type component associated with the member reference, giving:

```
01,
   02,
      03 DIM(2) FLOAT,
      03 CHAR(4),
   02 DIM(2,3) DEC(10)
```

   If this dimension were not moved, it would be deleted by Step 2c, and that would be inconsistent with the interpretation of an '*' subscript.

   c. The remainder of the storage type is edited. The component of the storage type that corresponds to the member reference, S1(I+3) is kept. Since S1 is not a structure, nothing else is kept. The result is:

```
03 DIM(2) FLOAT
```

   A level number on a storage type that is not a structure is not permitted, so the final result is

```
DIM(2) FLOAT
```

3. (Subscript Evaluation.) The subscript expression in the given reference is evaluated. The fully-bound reference is

```
Q(*).R1.S1(2)
```

4. (Value Retrieval.) An inspection of the 22 designators for the scalar components of Q shows that the fully-bound reference matches two, namely:

```
Q(1).R1.S1(2)
Q(2).R1.S1(2)
```

The sequence of two scalar values associated with these designators is retrieved and is the value of the given reference. Observe that the storage type of this result is identical to the storage type obtained in Step 2.

If the difference between two subscript expressions is ignored and if shortened references are excluded, then there are 18 distinct references to the variable designated by Q, as follows:

Q(*)	Q(I)
Q(*).R1	Q(I).R1
Q(*).R1.S1(*)	Q(I).R1.S1(*)
Q(*).R1.S1(J)	Q(I).R1.S1(J)
Q(*).R1.S2	Q(I).R1.S2
Q(*).R2(*,*)	Q(I).R2(*,*)
Q(*).R2(*,K)	Q(I).R2(*,K)
Q(*).R2(J,*)	Q(I).R2(J,*)
Q(*).R2(J,K)	Q(I).R2(J,K)

One of these forms of reference, Q(*).R1.S1(J), has just been considered in detail. Several other forms are now considered:

- Q(I).R1.S1(J) (assume I = 1 and J = 3)

  storage type: FLOAT

  designator: Q(1).R1.S1(3)

  (Out of the 18 forms of reference for Q, only three have scalar values. This is one of them.)

- Q(I).R2(J,K) (assume I = 1, J = 0, K = 2)

  storage type: DEC(10)

  designator: Q(1).R(0,2)

  (This is the second form that has a scalar value. The third is Q(I).R1.S2.)

- Q(*)

  storage type: 01 DIM(2),
  02,
  03 DIM(4) FLOAT,
  03 CHAR(4),
  02 DIM(2,3) DEC(10)

  designators: (The full sequence of 22 designators)

  (This reference designates the entire variable associated with Q.)

- Q(I).R1 (assume I = 2)

  storage type: 01,
  02 DIM(4) FLOAT,
  02 CHAR(4)

  designators: Q(2).R1.S1(1), Q(2).R1.S1(2),
  Q(2).R1.S1(3), Q(2).R1.S1(4),
  Q(2).R1.S2

  (This storage type is essentially the declaration of R1; only the level numbers have been changed.)

- Q(*).R1.S1(*)

  storage type: DIM(2,4) FLOAT

  designators: Q(1).R1.S1(1), Q(1).R1.S1(2),
  Q(1).R1.S1(3), Q(1).R1.S1(4),
  Q(2).R1.S1(1), Q(2).R1.S1(2),
  Q(2).R1.S1(3), Q(2).R1.S1(4)

  (Here, two separate dimensions combine to make a two-dimensional array.)

- Q(I).R2(*,*) (assume I = 2)

  storage type: DIM(2,3) DEC(10)

  designators: Q(2).R2(0,1), Q(2).R2(0,2), Q(2).R2(0,3),
  Q(2).R2(1,1), Q(2).R2(1,2), Q(2).R2(1,3)

  (Observe that the dimension of R2, which is '0:1', is normalized to 2; but this does not affect the subscripts used in the designators.)

- Q(*).R2(J,*) (assume J = 0)

  storage type: DIM(2,3) DEC(10)

  designators: Q(1).R2(0,1), Q(1).R2(0,2), Q(1).R2(0,3),
  Q(2).R2(0,1), Q(2).R2(0,2), Q(2).R2(0,3)

  (Here, the storage type is exactly the same as for the previous example; but the sequence of designators, and therefore the value, is different.)

Locator-Qualified Variable References

A locator-qualified variable reference makes use of a POINTER or OFFSET value to locate a variable. Once the variable is located, it can be accessed by any of the means thus far described in this discussion of variable references.

THE FORM OF A LOCATOR-QUALIFIED VARIABLE REFERENCE

A locator-qualified variable reference has the following form:

lq -> br

where lq is the locator qualifier and br is the based reference. The locator qualifier must be a reference that yields a POINTER or OFFSET value. The based reference must have the form of a simple variable reference, a subscripted variable reference, or a structure-qualified variable reference; however, the major name of the based reference must be declared BASED.

A simple example of a locator-qualified reference is:

```
P->X
```

In this example, P must be declared POINTER or OFFSET, and X must be declared BASED. In an English reading of a program, the reference can be expressed as "the X pointed to by P" or, more descriptively, as "the value obtained by interpreting the value of P as a pointer to a variable that has the storage type given by X".

Other examples of locator-qualified references are:

```
F(X+2,3*M)->Y

Q.ALPHA(J,K).R2->TOP(I+3).NEXT

B->G->H
```

In the first example, the locator qualifier is a function reference or a subscripted reference (depending on the declaration of F) that must be declared POINTER or OFFSET. In the second example, both the locator qualifier and the based reference are complicated structure-qualified references. In the third example, the locator qualifier for the entire reference is B->G, and the locator qualifier for B->G is B. For a given reference, it is always the rightmost arrow that separates the locator qualifier from the based reference.

ASSOCIATED STORAGE TYPES FOR LOCATOR VALUES

Every locator value has an associated storage type. This storage type is not the storage type of the locator value itself; that storage type is either POINTER or OFFSET. Instead, the associated storage type is the storage type of the variable that is designated by the locator value.

The associated storage type is created when the locator value is created. Two cases apply:

- A locator value is created as the result of the application of the ADDR built-in function to a given variable reference. In this case, the associated storage type is the storage type of the given variable reference.
- A locator value is created when an ALLOCATE statement is executed on a given variable name; the statement causes a locator value to be assigned (through a SET option) to a locator variable. In this case, the associated storage type is the storage type of the given variable name.

The associated storage type accompanies a locator value wherever it goes: as the value is assigned from one variable to another or as the value is returned by a programmed function reference.

As an example of the handling of associated data types, consider the following program:

```
P:   PROC;
        DCL (P1,P2,P3,P4) POINTER;
        DCL A(10) FLOAT;
        DCL 01 Q BASED,
               02 R1 FLOAT,
               02 R2 DEC(5);
        ...
        ALLOCATE Q SET(P1);
        ...
        P2 = ADDR(A);
        ...
        P3 = ADDR(A(3));
        ...
        P4 = P2;
        ...
        END;
```

After the ALLOCATE statement, the POINTER variable designated by P1 contains a pointer value; and that value has the associated storage type '01, 02 FLOAT, 02 DEC(5)'. After the first assignment statement, P2 has a pointer value whose associated storage type is DIM(10) FLOAT. After the second assignment statement, P3 has a pointer value whose associated storage type is FLOAT. After the last assignment statement, P4 has a pointer value whose associated storage type is DIM(10) FLOAT.

However, the PL/I compiler will accept and execution will not detect a locator reference in which the locator variable qualifies a variable of storage type different than its most recent associated storage type. As may be imagined, the absence of matching of the associated storage type of a pointer with the storage type of the variable pointed to can lead to program errors. Sometimes, of course, mismatching is done deliberately.

THE INTERPRETATION OF LOCATOR-QUALIFIED VARIABLE REFERENCES

A locator-qualified variable reference that is used for retrieval of a value is interpreted as follows:

1. Name Resolution. Resolve the major name in the based reference of the given reference. The result is the declaration of the major name.

2. Locator-Qualifier Evaluation. Evaluate the locator qualifier for the given reference. The result is a pointer value or an offset value; if it is an offset value, convert it to a pointer value. The result is the base pointer value for the given reference.

3. Base-Variable Location. Use the base pointer value to locate a position in storage. At that position, a variable begins whose storage type is the same as the associated storage type of the base pointer value. This variable is the base variable for the given reference.

4. Based-Variable Overlay. Overlay a based variable on the base variable. That is, define the set of designators that would have been produced if the base variable had been allocated in accordance with the declaration of the major name of the based reference.

5. Based Reference Interpretation. Interpret the based reference that occurs in the given reference. That is, determine its storage type, evaluate its subscripts, and evaluate it. The results are the storage type and value for the entire locator-qualified reference.

6. Based-Variable Withdrawal. Withdraw the based variable from the base variable; that is, discard the designators that were defined in Step 4.

Most of the interpretation is performed by the compiler; only locator-qualifier evaluation (Step 2) and a portion of based reference interpretation (Step 5) are performed during program execution.

According to Step 3 of the interpretation, the associated storage type of the base pointer matches the storage type of the major name in the based reference of the given locator-qualified reference. As noted above, however, a mismatch is not detected, and the interpretation simply assumes that the associated storage type of the locator variable does match the based reference storage type.

EXAMPLES OF LOCATOR-QUALIFIED VARIABLE REFERENCES

Examples of locator-qualified variable references are discussed here; they occur in the following program:

```
P:   PROC;
        DCL (N,K) FIXED;
        DCL 01 H(100) STATIC,
               02 FLAGS BIT(3),
               02 ITEM(2),
                  03 TEXT CHAR(6),
                  03 COUNT DEC(5),
               02 STYLE DEC(4);
        DCL 01 I(N) BASED,
               02 T CHAR(6),
               02 C DEC(5);
        DCL P1 POINTER;
        DCL SYSPRINT FILE;
        ... (assignments to variables occur here)
        PUT LIST(P1->I(2*K-4));
        ...
        END;
```

Just before the examples of retrieval are interpreted, suppose that storage includes the regions:

permanent internal region, example program

```
                                        1 1 1
17762   H  (1).FLAGS  "/1/1/0/"B

(... and so on for the first 7 elements of H)


                                        1 1 1
20144   H  (8).FLAGS  "/1/1/0/"B

                                              X X X X X X
20145   ----  .ITEM(1).TEXT  "/Q/N/A/I/L/S/"

                                          S 9 9 9 9 9
20147   ------------  .COUNT  /+/0/0/1/4/4/

                                          X X X X X X
20151   ---------  (2).TEXT  "/A/C/C/I/D/E/"

                                          S 9 9 9 9 9
20153   ------------  .COUNT  /+/0/0/0/9/0/

                                     S 9 9 9 9
20155   ----  .STYLE  /-/0/0/1/3/

(... and so on for the remaining 92 elements of H)
```

activation internal region, example program

```
                    ptr
01771   P1  / 20145 /
```

The interpretation of a locator-qualified reference is now given in detail. Consider the reference:

```
P1->I(2*K-4)      (assume K = 3, N = 2)
```

The steps in the interpretation are:

1. (Name Resolution.) The declaration of the major name, I, in the base reference is determined. It is given by the third DECLARE statement in the example program.

2. (Locator-Qualifier Evaluation.) The locator qualifier is the simple variable reference, P1; its value is the base pointer value, 20145. The associated storage type of the value must be the same as that declared for I. In fact, the only valid source for such a value would be ADDR(I).

3. (Base-Variable Location.) The base pointer value is used to locate a position in storage. Three variables begin at this position: H(8).ITEM(1).TEXT (a scalar), H(8).ITEM(1) (a structure), and H(8).ITEM (an array). The last of these has the same storage type as I, and it is therefore the base variable.

4. (Based-Variable Overlay.) The based variable is overlaid on the base variable. New designators are defined, and a portion of the diagram of storage is changed to read as follows:

```
                                                  1 1 1
20144              H  (8).FLAGS     "/0/0/0/"B

                                                  X X X X X X
20145  I(1).T  ----  .ITEM(1).TEXT  "/0/N/A/I/L/S/"

                                                  S 9 9 9 9 9
20147     .C  ----------  .COUNT    /+/0/0/1/4/4/

                                                  X X X X X X
20151  (2).T  ---------  (2).TEXT  "/A/C/C/I/D/E/"

                                                  S 9 9 9 9 9
20153     .C  ----------  .COUNT    /+/0/0/0/9/0/

                                                  S 9 9 9 9
20155              .STYLE           /-/0/0/1/3/
```

Observe that the variable I matches the base variable only because N is 2 at the time this step is performed; if N had any other value, the reference would be invalid.

5. (Reference Evaluation.) The subscript in the based reference is evaluated and the fully-bound reference is I(2). The result is:

```
storage type:     01,
                      02 CHAR(6),
                      02 DEC(5)

value:            "ACCIDE", 90
```

6. (Based-Variable Withdrawal.) The designators defined in Step 4 are discarded, and the diagram returns to the form that appeared at the beginning of this discussion.

Shortened Forms of References

The conventions for shortening variable references are described here, and specific guidelines for their use are given. A reference should be shortened only in order to make the program in which it occurs more clear to those who must read it. A reference should not be shortened merely to reduce the number of keystrokes required to type the program.

SUBSCRIPT-LIST DELETION

A subscripted reference can be shortened by deleting its subscript list, provided that all subscripts are '*' subscripts. Similarly, a structure-qualified reference can be shortened by deleting all of its subscript lists provided all of its subscripts are '*' subscripts.

Examples of Subscript List Deletion

Examples of the shortening of a reference by deletion of subscript lists are:

Reference	Shortened Reference
A(*)	A
B(*,*)	B
C(*).D.E(*,*)	C.D.E

Observe that the third reference, C(*).D.E(*,*), cannot be shortened to C(*).D.E or C.D.E(*,*); if any subscript list is deleted, all must be.

Guidelines for Subscript List Deletion

The deletion of a subscript list is not recommended. Any reference with an '*' is necessarily a reference to an aggregate value. The PL/I facilities for handling aggregates are expensive and should not be used casually. Indeed, the presence of a subscript list composed of asterisks is a useful warning that an aggregate value is being processed.

NAME DELETION

A structure-qualified reference can be shortened by deleting one or more of its containing references, provided that the deleted references are unsubscripted names and provided that the deletion does not change the declaration of the reference.

The declaration of a reference is changed if the unshortened reference is governed by one declaration and the shortened reference is governed by another. The declaration that governs the reference is determined by rules given earlier in the section on "Declarations".

A subscript list can be moved within a reference. The purpose of this convention is to allow deletion of a containing reference which, aside from its subscript list, satisfies the conditions for deletion. For example, the reference X(I+2).Y can be written as X.Y(I+2) and then, If the declaration does not change, as Y(I+2).

Examples of Name Deletion

The following program is used to illustrate the deletion of containing names to produce shortened references:

```
P:   PROC;
        DCL 01 VEHICLE,
               02 SERIAL DEC(10),
               02 COST DEC(8,2);
        ...
        BEGIN;
             DCL 01 SALE,
                    02 CUSTOMER,
                       03 NAME CHAR(30),
                       03 ADDRESS CHAR(60),
                    02 SUPPLIER,
                       03 NAME CHAR(8),
                       03 COST(2) DEC(10,2);
             DCL COST DEC(10,2);
             ...
             ... (example references occur here)
             ...
             END;
     END;
```

Observe that NAME is declared twice and COST is declared three times. These declarations are all valid but they make certain shortened references invalid.

Assume that the declarations explicitly shown are the only declarations in the program. Then the references that could be used in the inner block are classified as follows:

Unshortened References	Valid Shortening	Invalid Shortening
VEHICLE.SERIAL	SERIAL	
VEHICLE.COST		COST
SALE.CUSTOMER	CUSTOMER	
SALE.CUSTOMER.NAME	CUSTOMER.NAME	SALE.NAME NAME
SALE.CUSTOMER.ADDRESS	CUSTOMER.ADDRESS SALE.ADDRESS ADDRESS	
SALE.SUPPLIER.NAME	SUPPLIER.NAME	SALE.NAME NAME
SALE.SUPPLIER.COST(I)	SUPPLIER.COST(I) SALE.COST(I)	COST(I)

The interesting references are the "invalid shortenings". They are accounted for as follows:

- The references SALE.NAME and NAME are invalid because they are ambiguous. Each is a partially-qualified reference to both SALE.CUSTOMER.NAME and SALE.SUPPLIER.NAME.

- The reference COST is an invalid shortening of VEHICLE.COST because (in the inner block) it would be interpreted as a reference to the COST declared in the third DECLARE statement.

- The reference COST(I) is an invalid shortening of SALE.SUPPLIER.COST(I) because (since subscripts are ignored in the resolution of a reference) it would also be interpreted as a reference to the COST declared in the third DECLARE statement.

Guidelines for Name Deletion

The deletion of the leftmost reference of a structure-qualified variable reference is not recommended, even if a careful analysis shows that the result is correct. The leftmost level reference is much more important than the other level references because it determines which major variable is being referenced.

The moving of a subscript list from one level reference to another within a structure-qualified reference is not recommended. Although it has no effect on the interpretation of the reference by the processor, it gives the human reader the wrong storage type for the reference. The deletion of containing references should be confined to those that are originally unsubscripted.

LOCATOR-QUALIFIER DELETION

A locator-qualified reference can be shortened by deleting the locator-qualifier, provided that the major name of the based reference is declared with the attribute

BASED(x)

where x is the locator-qualifier that occurs in the unshortened reference.

Guidelines for Locator-Qualifier Deletion

The deletion of a locator-qualifier is not recommended for most applications. The use of based variables is an error-prone aspect of PL/I programming, and the deletion of a locator-qualifier introduces additional possibilities for errors.

The Cost of Variable References

The relative complexity of the various kinds of variable references is a bad guide to the cost of these references. A PL/I program is compiled, and the cost of the interpretation of any construct, and references in particular, is divided between compilation and execution. Whatever can be performed during compilation becomes a negligible cost because it is performed only once for each compilation of the program.

The cost of interpreting a reference consists of a cost that is approximately the same for all variable references plus the cost of evaluating any expressions (subscripts or a locator qualifier) that occur in the reference. A long and complicated reference that contains only constant expressions, such as

```
ALPHA(3).BETA.GAMMA(5,2)
```

costs no more to interpret than a simple variable reference. Generally speaking, the organization of data into structures and the corresponding use of structure-qualified variables do not, in themselves, increase the cost of referencing the data.

CONSTANT LITERALS

A constant literal designates a computational constant value. The constant literal gives, in the spelling of the construct itself, both the data type and the value of the constant it designates. For example, the constant literal 23.9 designates a constant whose storage type is REAL FIXED DECIMAL (3,1) and whose value is 23.9. The spelling of a constant literal is almost, but not quite, the same as the stored value representation it designates. For example, the constant literal 23.9 has the stored value representation '+23.9'.

A constant literal designates a constant that requires at most 64 words of storage (for the longest possible character-string constant). Furthermore, the value of a constant literal can always be determined at compilation time, and various optimization techniques can be applied to reduce the cost of its storage. For these reasons, the allocation and initialization of storage for the value designated by a constant literal need not be described. It is sufficient to show how, for a given constant literal, the storage type and value of the literal can be determined.

Arithmetic Constant Literals

The language provides a full range of arithmetic constant literals, including both FIXED and FLOAT scaling, DECIMAL and BINARY base, and REAL and COMPLEX mode.

THE FORM OF ARITHMETIC CONSTANT LITERALS

The form of the arithmetic constant literals is given by the following rules:

1. An integer literal is a sequence of one or more digits.

2. A fixed literal is either an integer literal or is an integer literal modified by the insertion of a decimal point before or after any digit.

3. A float literal is a fixed literal (called the mantissa) followed by an E followed by a signed integer literal (called the exponent).

4. A decimal literal is any fixed or float literal. The exponent of a float decimal is considered to be a power of ten.

5. A binary literal is a fixed or float literal followed by a B. Except for an exponent, the literal must contain only binary digits. The exponent of a float binary literal is interpreted as a decimal number; it is considered to be a power of two.

6. A real literal is any decimal or binary literal.

7. An imaginary literal is any decimal or binary literal followed by an I.

8. An arithmetic literal is any real or imaginary literal.

An arithmetic literal must not be more than 256 characters long.

The rules just given build on one another. An integer literal is used to build a fixed literal, a fixed literal is used to build a float literal, and so on. The following shows how the rules are used to build the literals '11101110B' and '8.2300E-3I':

1.	integer	11101110	82300
2.	fixed	11101110	8.2300
3.	float	---	8.2300E-3
4.	decimal	---	8.2300E-3
5.	binary	11101110B	---
6.	real	11101110B	8.2300E-3
7.	imaginary	---	8.2300E-3I
8.	arithmetic	11101110B	8.2300E-3I

THE INTERPRETATION OF ARITHMETIC CONSTANT LITERALS

The interpretation of an arithmetic constant literal must yield a storage type and a value. The storage type is determined as follows:

- The aggregate type is always scalar.
- The mode is COMPLEX if the reference ends with an I, and is REAL otherwise.
- The scaling is FLOAT if the reference has an E followed by an exponent and is FIXED otherwise.
- The base is BINARY if the reference contains a B and is DECIMAL otherwise.
- The number-of-digits in the precision is obtained by counting all the digits except those in an exponent.
- The scale-factor in the precision is obtained by counting the digits to the right of the point except for those in an exponent. If there is no point, the scale-factor is zero.

The value of the constant literal is the value represented by the spelling of the literal.

EXAMPLES OF ARITHMETIC CONSTANT LITERALS

Examples of arithmetic constant literals follow. Each example is accompanied by its storage type and its representation in storage.

Constant Literal	Storage Type	Representation
23.9	REAL FIXED DECIMAL(3,1)	+23.9
0	REAL FIXED DECIMAL(1,0)	+0.
000	REAL FIXED DECIMAL(3,0)	+000.
110.011011101B	REAL FIXED BINARY(12,9)	+110.011011101B
0B	REAL FIXED BINARY(1,0)	+0.B
5.8000000E3	REAL FLOAT DECIMAL(8)	+58000000.E-4
.1110010011101E-3B	REAL FLOAT BINARY(13)	+.1110010011101E-3B
568E0I	COMPLEX FLOAT DECIMAL(3)	+000.E0+568.E0I
9.000E5I	COMPLEX FLOAT DECIMAL(4)	+0000.E0+9000.E2I
10101110101BI	COMPLEX FIXED BINARY(11,0)	+10101110101.BI

GUIDELINES FOR ARITHMETIC CONSTANT LITERALS

The storage type of an arithmetic constant literal should be chosen for convenience and clarity. Use FIXED scale unless the magnitude of the values requires an exponent for convenience of representation. Use DECIMAL base except in the very rare case that the problem formulation depends on binary arithmetic. Use COMPLEX mode if the value has an imaginary part.

Consider, for example, the assignment of a constant to a variable. Suppose that variable, X, is declared as follows:

```
DCL X FLOAT;
```

so that its storage type is FLOAT BIN(27). In order to assign the value 24 to this variable, it might be thought appropriate to use a FLOAT BIN(27) literal constant and write:

```
X = 00000000000000000000011000E0B
```

However, this statement is inconvenient and unclear. Instead, it is better to write:

```
X = 24;
```

The necessary conversion from FIXED DEC(2) to FLOAT BIN(27) is performed at negligible cost during compilation; therefore no conversion is required when the statement is executed.

String Constant Literals

The language provides constant literals for both character-string and bit-string values. Special conventions permit the use of any ASCII character within a character-string literal, and thus provide complete generality in string manipulation.

THE FORM OF STRING CONSTANT LITERALS

The form of the string constant literals is given by the following rules:

1. A character-string literal is an optional replication factor (defined by Rule 3), followed by a double-quote character, followed by a sequence of zero or more ASCII characters, followed by a double-quote character. The character sequence between the double-quote characters represents the value of the string; however, two double-quote characters must appear in the sequence for each double-quote character in the value.

2. A bit-string literal is an optional replication factor, followed by a double-quote character, followed by a sequence of zero or more 0 and 1 characters, followed by a double-quote character, followed by a B.

3. A replication factor is a parenthesized integer literal whose value is greater than zero. Suppose the value of the replication factor for a given reference is n and that the sequence of characters between the double-quote characters is s. Then an equivalent string literal is obtained by replacing s with n copies of s and deleting the replication factor.

A string literal must not be more than 256 characters long. If a string literal has a replication factor, the restriction on length is applied to the equivalent literal that does not have a replication factor. For example, (257)"0"B is considered to be a literal of 257 characters and therefore is invalid.

THE INTERPRETATION OF STRING CONSTANT LITERALS

The interpretation of a string constant literal must yield a storage type and a value. The storage type is determined as follows:

- The aggregate type is always scalar.
- The literal is BIT(n) or CHARACTER(n) depending on whether a B occurs at the end or not. The n is the number of characters in the sequence between the double-quote characters; a pair of double-quote characters in the sequence counts as one character.
- The literal is always NONVARYING.

The value of the literal is the sequence between double-quote characters with the provision, already noted, that two double-quote characters in the sequence represent one double-quote character in the value.

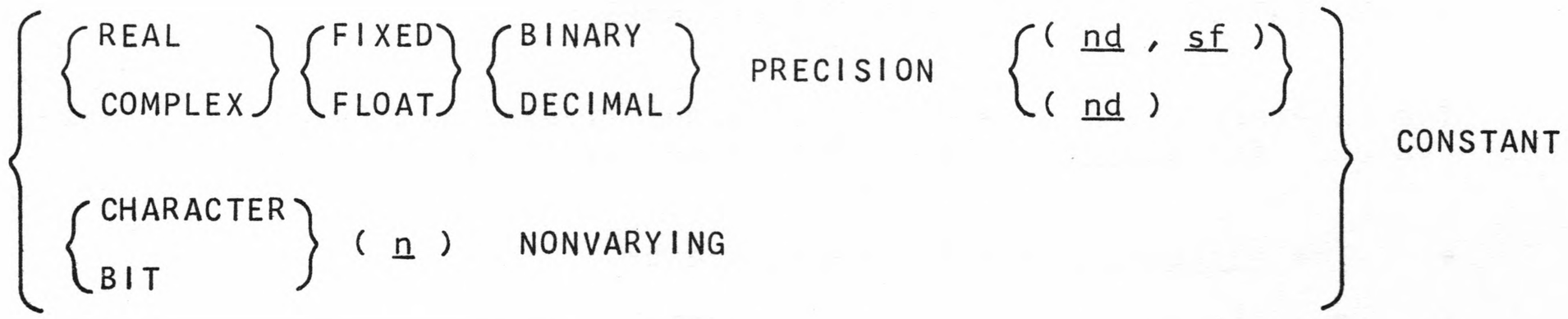
EXAMPLES OF STRING CONSTANT LITERALS

Examples of string constant literals follow:

Constant Literal	Storage Type	Value Representation
"SAY NOTHING."	CHARACTER(12) NONVARYING	"SAY NOTHING."
"SAY ""STOP!""."	CHARACTER(12) NONVARYING	"SAY "STOP!"."
""	CHARACTER(0) NONVARYING	""
"100011"B	BIT(6) NONVARYING	"100011"B
"1"B	BIT(1) NONVARYING	"1"B
""B	BIT(0) NONVARYING	""B
(2)"MOSHE "	CHARACTER(12) NONVARYING	"MOSHE MOSHE "
(12)"1"B	BIT(12) NONVARYING	"111111111111"B

Attributes for Constant Literals

The complete attribute set for any constant literal is given by the following diagram:



In this diagram,

- nd (number of digits) is an unsigned integer
- sf (scale factor) is an optionally signed integer
- n (maximum length) is an unsigned integer

The attribute set for a constant literal is never written in a program; but it must be determined as part of the interpretation of the program.

The Cost of Constant Expressions

Constant expressions are evaluated at execution time by instructions generated by the PL/I compiler instead of by the compiler itself. In general, then, despite their worth for documentation, constant expressions should be evaluated by the programmer and the resulting constant written in place of the expression (which may be relegated to a comment).

CONSTANT REFERENCES

A constant reference designates a constant statement address value or a constant FILE value. A constant reference can appear in two kinds of context. The most common context is one that makes final use of the value of the constant reference; for example, a LABEL constant reference in a GOTO statement provides the destination for transfer of control. The second context is one that saves the value of the constant reference for later use; for example, the assignment of the value of a LABEL constant reference to a LABEL variable or the use of a LABEL constant reference as an argument in a function reference.

Observe that the range of constant references is limited. Except for LABEL constant references, they handle only scalar values. No provision is made for constant references for POINTER, OFFSET, or AREA values.

Statement Constant References

A statement constant reference can appear wherever a statement address value is required. The most common use of a statement constant reference is in a context that makes final use of the value of the reference. For a LABEL constant reference, this context is a GOTO statement; for an ENTRY constant reference, it is a CALL statement or a function reference; and for a FORMAT statement, it is the R format item that is used in connection with edit-directed stream input/output.

THE FORM OF STATEMENT CONSTANT REFERENCES

A statement constant reference has one of the following forms:

scn

scn(se)

where scn is the statement constant name and se is the subscript expression. The statement constant name must be an identifier that has the type attribute LABEL, ENTRY, or FORMAT. The second form can be used only with a LABEL constant name. The subscript expression must yield a value that can be converted to an integer.

THE INTERPRETATION OF STATEMENT CONSTANT REFERENCES

The evaluation of a statement constant reference yields a LABEL value, an ENTRY value, or a FORMAT value. Such a value contains a statement designator and an activation index. The activation index has no significance except in a program that uses general recursion; it is described later, at the end of the section on "Procedure Invocation".

The statement designator that is part of the value of a statement constant reference is defined by the label prefix that declares the identifier in the statement constant reference. For example, the statement constant reference ALPHA designates a statement that (1) has the label prefix 'ALPHA:' and (2) is contained in the smallest block that contains both the given reference and a statement with the label prefix 'ALPHA:'.

EXAMPLES OF STATEMENT CONSTANT REFERENCES

As a simple example of the use of a statement constant reference, consider the following program:

```
P1:  PROC;
LAB:    CALL SR(X);
        ...
        GOTO LAB;
        ...
        END;
```

The occurrence of 'LAB:' at the beginning of the CALL statement is the defining label prefix for LAB. By virtue of that prefix, LAB is declared LABEL INTERNAL and is given the address of the CALL statement as its value. The occurrence of LAB in the GOTO statement is a LABEL constant reference.

As a more general example of the use of statement address constant references, consider:

```
P2:  PROC;
        ... (Computation #1)
        CALL M3;
        PUT EDIT( ... )(R(F));
M3:     PROC;
             ... (Computation #2)
             GOTO L(I+2);
L(1):        ... (Computation #3)
             GOTO A;
L(-1):       ... (Computation #4)
             GOTO A;
L(2):
A:           ... (Computation #5)
             END;
F:      FORMAT( ... );
        END;
```

In the CALL statement, M3 is an ENTRY constant reference; in the PUT statement, F is a FORMAT constant reference; and in the GOTO statement, L(I+2) is a LABEL constant reference.

The example program performs Computation #1, calls the procedure M3, and executes the PUT statement. When the procedure M3 is called, it performs Computation #2 and then proceeds as follows:

- If I+2=1, it performs Computation #3 and Computation #5.
- If I+2=-1, it performs Computation #4 and Computation #5.
- If I+2=2, it performs Computation #5.
- If I+2=0, execution is undefined.
- If I+2 is not in the range -1 through 2, the SUBSCRIPTRANGE condition occurs.

When the PUT statement is executed, a reference is made to the FORMAT statement, and that statement supplies the format items for the output.

EXTERNAL ENTRY CONSTANT REFERENCES

There is one case in which a label prefix is not sufficient declaration for a statement constant name; this case arises when an external ENTRY constant is defined in one external procedure and is used in another external procedure. In the defining procedure, the constant is declared by a label prefix as already described. However, in other external procedures that refer to the ENTRY constant, the constant name must be declared again; that is, the name may be declared with the EXTERNAL and ENTRY attributes by means of a DECLARE statement or by a CALL reference, since the PL/I compiler provides a default ENTRY typing of EXTERNAL if the name is not declared as an internal procedure or entry.

If no ENTRY declaration is given for a CALL reference, however, the PL/I compiler generates a less efficient calling sequence for each call with arguments to an external entry which identifies the type and precision of each argument. Hence, to avoid the less efficient calling sequence, the programmer is advised to supply an ENTRY declaration if there are arguments.

When an ENTRY declaration is given that specifies type and precision for each argument, the PL/I compiler generates instructions to convert any argument of a different type or precision. As noted below, if the entry name is used in an external function reference, an ENTRY declaration is required to specify the type and precision of the result returned by the function.

As an example of the declaration of external ENTRY constant names, consider the following program:

```
P1:  PROC;
        DCL P2 ENTRY(FLOAT,DEC(10));
        DCL P3 ENTRY(FLOAT);
        ...
        CALL P2(X,Y);
        ...
        CALL P3(Z);
        ...
        END;

P2:  PROC(R,S);
        DCL R FLOAT;
        DCL S DEC(10);
        ...
P3:     ENTRY(Q);
        DCL Q FLOAT;
        ...
        END;
```

This program is made up of two external procedures. The declarations of P2 and P3 in the second external procedure are provided by label prefixes. Because these names are used in the first external procedure, they are declared in that procedure by means of DECLARE statements.

File Constant References

A file constant reference can appear wherever a file value is required. The most common use of a file constant reference is in the context that makes final use of the value of the reference; that is, a FILE option in a statement that performs input/output or opens or closes a file.

THE FORM OF FILE CONSTANT REFERENCES

A file constant reference has the following form:

fcn

where *fcn* is a *file constant name*. A file constant name is an identifier that is an identifier that is declared with the attribute FILE.

EXAMPLES OF FILE CONSTANT REFERENCES

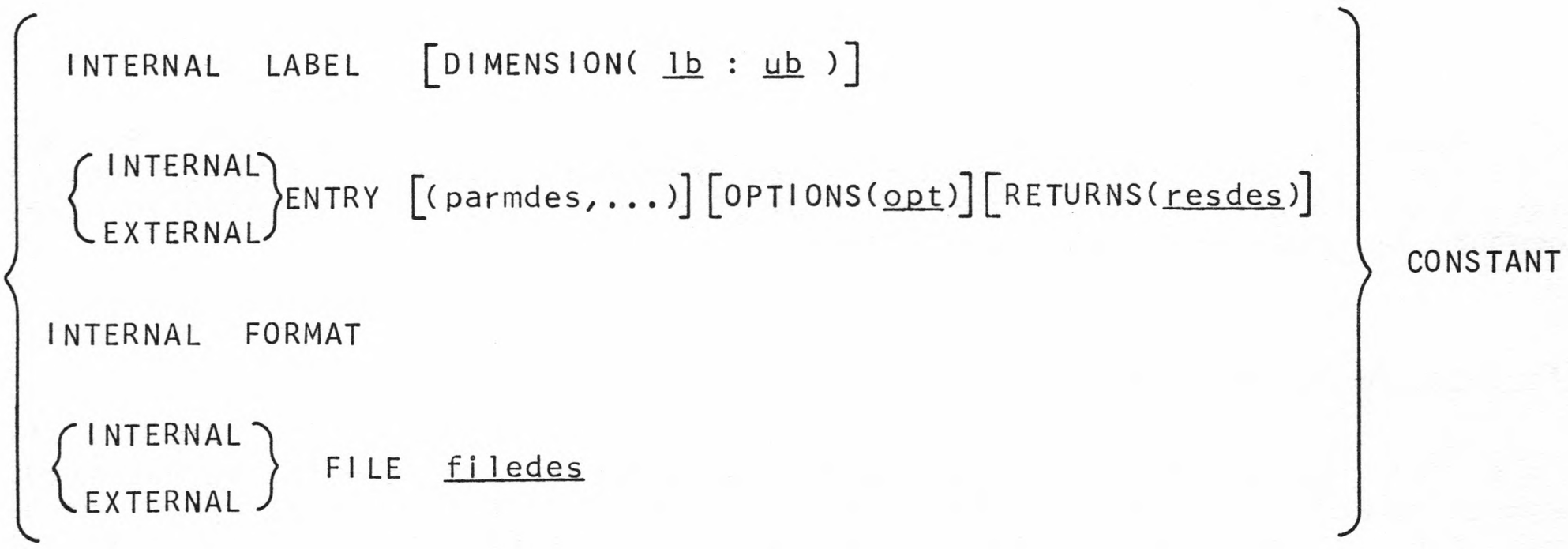
As an example of the use of file constant references, consider the following program:

```
P:   PROC;
        DCL (ALPHA,BETA) FILE;
        ...
        OPEN FILE(ALPHA) INPUT STREAM;
        OPEN FILE(BETA) OUTPUT PRINT STREAM;
        ...
        GET FILE(ALPHA) LIST(A,B,C);
        ...
        PUT FILE(BETA) LIST(X,Y,Z);
        ...
        CLOSE FILE(ALPHA);
        CLOSE FILE(BETA);
        ...
        END;
```

This program shows how the file constant references ALPHA and BETA are used to designate file-state blocks for an input data set and an output data set, respectively.

Attributes for Constant Names

The complete attribute set for named constants is given by the following diagram:



In this diagram,

- lb (lower bound) and ub (upper bound) are optionally-signed integers.
- parmdes is a parameter descriptor and resdes is a result descriptor; these constructs are described later, in the section on "Procedure Invocation".
- opt is an OPTIONS type for external entries as described in the section on "Procedure Invocation".
- filedes is a file description; this construct is described later, in the sections on "Stream Input/Output" and "Record Input/Output".

The attribute set for a LABEL or FORMAT constant name is never written in a program; instead, it is deduced from a label prefix. The attribute set for an entry constant name is written in a DECLARE statement only in an external procedure that uses but does not define the entry constant; otherwise, it is deduced from a label prefix. The attribute set for a file constant name is always given in a DECLARE statement.

PROGRAMMED FUNCTION REFERENCES

A programmed function reference invokes a PL/I procedure and then delivers the result of the execution of the procedure as the value of the reference. Through the use of a programmed function reference, a long and complicated procedure can be executed in the midst of the evaluation of an expression.

The Form of Programmed Function References

A programmed function reference has one of the following forms:

ref(arglist)

ref( )

where ref is the entry reference and arglist is the argument list. Usually, the entry reference is an entry constant name; however, it can also be a generic entry name or any reference that yields a scalar entry value. The argument list is a sequence of arguments separated by commas, and each argument is an expression. The second form is used when no arguments are required.

A generic entry name does not directly designate an ENTRY value; instead, it is replaced by an entry constant name during the compilation of the program. The declaration of the generic entry name gives a set of entry name constants. For each entry name constant in the declaration, some attributes for each argument of the designated procedure entry point are given. Thus when a programmed function reference begins with a generic function name, the interpretation of the generic entry name is determined by the storage types of the arguments of the programmed function reference. An example of the use of a generic entry name is given in the section on "Procedure Invocation".

The Interpretation of Programmed Function References

The interpretation of programmed function references is fully described later, in the section on "Procedure Invocation". That interpretation can be summarized in two steps, as follows:

1. Storage Type Determination. Obtain the storage type of the entry reference of the given programmed function reference. This storage type includes an ENTRY attribute and a RETURNS attribute, and these attributes provide information about the procedure entry point that is designated by the entry reference. The ENTRY attribute gives a storage type for each parameter of the designated procedure entry point; this information is used in interpreting the argument. The RETURNS attribute gives the storage type of the value returned by the designated entry point and thus gives the storage type of the given programmed function reference.

2. Reference Evaluation. Determine the value of the programmed function reference. This requires the evaluation of the entry reference; the interpretation of the arguments; the activation, execution, and exit from the procedure; and the retrieval of the result of the procedure.

Step 1 of this interpretation is performed by the compiler before the program is executed. Step 2 is performed each time the programmed function reference is evaluated during program execution.

Examples of Programmed Function References

As a simple example of the use of programmed function references, consider the following program:

```
P:    PROC;
         DCL (SYSIN,SYSPRINT) FILE;
         DCL (A,B,C) FLOAT;
         GET LIST(A,B);
         C = F(A) + F(B);
         PUT LIST(C);
F:       PROC(X) RETURNS(FLOAT);
              DCL X FLOAT;
              IF X < 0
                   THEN RETURN(0);
                   ELSE RETURN(X);
              END;
         END;
```

In this program, the procedure F is especially simple; its result is either zero or the given argument value depending on whether the argument is negative or nonnegative. The important point, however, is that the definition of F is a separate part of the program and can be examined and modified separately.

In order to execute the example program, it is necessary to interpret the following operator expression:

```
F(A) + F(B)
```

In order to interpret this expression, it is necessary to know the storage type of the two programmed function references that appear in it. Consider the way in which the storage type of F(A) is determined. First, the storage type of the entry reference, F, is determined; it is:

```
ENTRY(FLOAT) RETURNS(FLOAT)
```

This storage type is obtained by applying the rules for the interpretation of an ENTRY constant reference to F; those rules are given earlier in this section. The storage type of F(A) is obtained from the RETURNS attribute in the storage type of F; it is:

```
FLOAT
```

In the same way, it can be shown that the storage type of F(B) is also FLOAT.

A second example of a programmed function reference follows. This example uses both a variable reference and a programmed function reference as the entry reference of a programmed function reference. Such usage occurs only in large and complicated programs, and a short example cannot be realistic; however, the example is formally correct and is used to show how the storage type of a complicated entry reference is determined.

```
P:    PROC;
         DCL (SYSIN,SYSPRINT) FILE;
         DCL (M,N) FIXED;
         DCL X FLOAT;
         DCL FV ENTRY(FIXED) RETURNS(ENTRY(FLOAT) RETURNS(FLOAT)) VARIABLE;
         GET LIST(M,N,X);
         IF M=0 THEN FV = F1; ELSE FV = F2;
         PUT LIST(FV(N)(X));
F1:      PROC(A) RETURNS(ENTRY(FLOAT) RETURNS(FLOAT));
              DCL A FIXED;
              IF A=0 THEN RETURN(F3); ELSE RETURN(F4);
              END;
F2:      PROC(B) RETURNS(ENTRY(FLOAT) RETURNS(FLOAT));
              DCL B FIXED;
              IF B=0 THEN RETURN(F4); ELSE RETURN(F3);
              END;
F3:      PROC(Z1) RETURNS(FLOAT);
              DCL Z1 FLOAT;
              RETURN(SIN(Z1));
              END;
F4:      PROC (Z2) RETURNS (FLOAT);
              DCL Z2 FLOAT;
              RETURN(SIN(Z2));
              END;
         END;
```

The central feature of the program is the interpretation of the programmed function reference:

```
FV(N)(X)
```

By the time this reference is evaluated, one of the ENTRY constant references F1 or F2 has been assigned to the ENTRY variable named FV. Thus the given reference is equivalent to one of the following:

```
F1(N)(X)          (for M = 0)

F2(N)(X)          (for M ≠ 0)
```

The evaluation of the programmed function reference F1(N) yields the ENTRY value designated by either F3 or F4, depending on whether N is zero or not. Similarly, the evaluation of F2(N) yields the ENTRY value designated by either F4 or F3, depending on whether N is zero or not. Thus, finally, the given reference is equivalent to one of the following:

```
F3(X)             (for M and N both zero or both nonzero)

F4(X)             (for other values of M and N)
```

Clearly this example could be programmed in a simpler and more efficient way, but it would not then illustrate the use of a non-constant entry reference.

The storage type of FV(N)(X) can be obtained before program execution begins, as follows. First, the storage type of the variable FV is determined from the DECLARE statement to be:

```
ENTRY(FIXED) RETURNS(ENTRY(FLOAT) RETURNS(FLOAT))
```

This storage type means that the value of FV is "an ENTRY value that designates a procedure entry point that has a FIXED parameter and returns an ENTRY value; and the latter ENTRY value designates a procedure entry point that has a FLOAT parameter and returns a FLOAT value". Next, the storage type of the programmed function reference FV(N) is determined from the storage type of FV to be:

```
ENTRY(FLOAT) RETURNS(FLOAT)
```

This storage type means that the value of FV(N) is "an ENTRY value that designates a procedure entry point that has a FLOAT parameter and returns a FLOAT value". Finally, the storage type of the entire reference is determined from the storage type of FV(N) to be:

```
FLOAT
```

This storage type means, of course, that the value of FV(N)(X) is "a FLOAT value".

BUILT-IN FUNCTION REFERENCES

A built-in function reference performs a specific calculation on its arguments and delivers the result as the value of the reference. For each built-in function, the calculation performed is part of the definition of PL/I. Although a built-in function reference resembles a programmed function reference in some ways, there are important and fundamental differences between the two kinds of reference. These differences are discussed here, after the form and interpretation of built-in function references are given.

A specific definition for each of the built-in functions is given in the next section, "Operations". Each definition gives restrictions on the arguments and gives rules for converting the arguments, determining the storage type of the result, and calculating the value of the result. The section on "Operations" is large, and the quickest way to find the definition of a particular built-in function is to look its name up in the index of this manual.

The Form of Built-in Function References

A built-in function reference has one of the following forms:

bifname(arglist)

bifname( )

bifname

where bifname is the built-in function name and arglist is the argument list. If referenced without a parenthesized arglist, the built-in function name must be declared BUILTIN and must be one of the following identifiers:

ABS	CEIL	DIMENSION	LINENO	ONLOC	SIZE
ADD	CHARACTER	DIM	LOG	ONSOURCE	SQRT
ADDR	CHAR	DIVIDE	LOG10	PAGENO	
ADDREL	COLLATE	DOT	LOG2	POINTER	STRING
AFTER	COMPLEX	EMPTY	LOW	PTR	SUBSTR
ALLOCATION	CPLX	ERF	MAX	PRECISION	SUBSTRACT
ALLOC	CONJG	ERFC	MIN	PREC	SUM
ATAN	CONVERT	EXP	MOD	PROC	TAN
ATAND	COPY	FIXED	MULTIPLY	REAL	TAND
ATANH	COS	FLOAT	NULL		TANH
	COSD	FLOOR	NULLO	REVERSE	TIME
	COSH	HBOUND	OFFSET	ROUND	TRANSLATE
BEFORE	DATE	HIGH	ONCHAR	SEARCH	TRUNC
BINARY	DECAT	IMAG	ONCODE	SIGN	UNSPEC
BIN	DECIMAL	INDEX	ONFIELD	SIN	VALID
BIT	DEC	LBOUND	ONFILE	SIND	VERIFY
BOOL		LENGTH	ONKEY	SINH	

The argument list is a sequence of expressions separated by commas. The second and third forms of a built-in function reference are equivalent; either can be used for a built-in function that requires no arguments.

The Interpretation of Built-in Function References

A programmed function reference is interpreted in two steps, as follows:

1. Storage Type Determination. Determine the target storage type for each argument and the storage type of the result. The rules for determining these storage types are given in the individual definitions of the built-in functions; often they depend on the storage types of the arguments.

2. Reference Evaluation. Evaluate each argument and convert it to the target storage type for the argument. Evaluate the reference as specified in the definition of the built-in function.

Step 1 of this interpretation is performed by the compiler before the program is executed. Step 2 is performed each time the built-in function reference is evaluated during program execution.

Examples of Built-in Function References

As an example of a built-in function reference that handles storage types in a simple but quite typical way, consider the following use of the SIN built-in function:

```
DCL (Y,ALPHA) FLOAT;
...
Y = SIN(ALPHA);
```

The function reference under consideration is:

```
SIN(ALPHA)
```

The built-in function name is SIN and the argument list contains one argument, ALPHA.

The storage type of the argument, ALPHA, of the built-in function reference in the example is:

```
REAL FLOAT BINARY(27)
```

According to the definition of SIN in the section on "Operations", the target storage type is:

```
REAL FLOAT BINARY(27)
```

Thus for this use of SIN, the argument is not converted before the calculation begins. Also according to the definition of SIN, the result storage type is:

```
REAL FLOAT BINARY(27)
```

Thus the storage type of the result is the same as that of the argument. When the assignment statement is executed, the value of ALPHA is fetched, the sine is calculated, and the result is returned as the value of the reference.

As an example of a more complicated built-in function reference, consider the following use of the MAX built-in function:

```
DCL X FLOAT;
DCL A FLOAT;
DCL B FIXED(35);
...
X = MAX(A,B,200);
```

Here, the assignment statement assigns the largest of the values designated by A, B, and 200 to the variable named X. A precise understanding of the assignment statement requires the determination of the storage type of the result of the reference to MAX.

The following table gives the argument storage types and the target storage types for the built-in function reference:

Argument	Argument Storage Type	Target Storage Type
A	REAL FIXED BINARY(27)	REAL FLOAT BINARY(27)
B	REAL FIXED BINARY(35)	REAL FLOAT BINARY(35)
200	REAL FIXED DECIMAL(3)	REAL FLOAT BINARY(10)

Observe that the target storage types differ only in the precision attribute; the choice of FLOAT over FIXED and of BINARY over DECIMAL is a rule that applies to many built-in functions. The storage type of the result is:

```
REAL FLOAT BINARY(35)
```

Observe that the number-of-digits is the maximum of those given in the target storage types of the arguments.

As an example of a built-in function reference with aggregate arguments, consider the following use of the MAX and MIN built-in functions:

```
DCL 01 ALPHA(3),
       02 A FLOAT,
       02 B FIXED;
DCL 01 TOP,
       02 A FLOAT,
       02 B FIXED;
...
ALPHA = MIN(MAX(0,ALPHA),TOP);
```

Here, the assignment statement changes the value of ALPHA where necessary so that its components lie in the range:

$$0 \leq \text{ALPHA.A(I)} \leq \text{TOP.A}$$

$$0 \leq \text{ALPHA.B(I)} \leq \text{TOP.B}$$

(for I = 1, 2, and 3)

When a value is changed, it is changed as little as possible; for example, a negative value is changed to zero.

The following table gives the argument storage types and the target storage types for the built-in function references:

Argument	Argument Storage Type	Target Storage Type
0	REAL FIXED DEC(1)	01 DIM(1:3), 02 REAL FLOAT BIN(4), 02 REAL FIXED BIN(4)
ALPHA	01 DIM(1:3), 02 REAL FLOAT BIN(27), 02 REAL FIXED BIN(17)	01 DIM(1:3), 02 REAL FLOAT BIN(27), 02 REAL FIXED BIN(17)
TOP	01, 02 REAL FLOAT BIN(27), 02 REAL FIXED BIN(17)	01 DIM(1:3), 02 REAL FLOAT BIN(27), 02 REAL FIXED BIN(17)

Observe that the scalar, 0, and the structure, TOP, are converted to the aggregate type of ALPHA.

Differences Between Built-in and Programmed Function References

The essential difference between built-in and programmed function references is in the way the actions performed are defined. For a built-in function reference, the action is defined as part of the PL/I language, and a given built-in function name means the same thing wherever it is used. In contrast, for a programmed function reference, the action is defined as part of a program, and a given programmed function name can mean different things under different circumstances.

In addition to this essential difference, there are several other important differences between built-in and programmed function references. These differences are:

1. A built-in function reference must begin with a built-in function name; therefore, the selection of the built-in function is made when the program is written. In contrast, a programmed function reference can, when necessary, begin with an entry variable reference or an entry programmed function reference; therefore, the selection of the programmed function can be made as part of each evaluation of the programmed function reference, and can change from one evaluation to the next. A built-in function name cannot be assigned as the value of an entry variable.

2. A specific argument of a specific built-in function reference can have any of several storage types without undergoing conversion of its storage type. In contrast, unless a generic function name is used, an argument of a programmed function reference must have a specific storage type (except for variations in extents) in order to escape conversion.

3. The result of a built-in function reference has a storage type that is derived from the storage type of its arguments. In contrast, the result of a programmed function reference is independent of the storage types of its arguments (except, perhaps for extents) and is determined by the definition of the function.

4. On a less important level, the parentheses around the argument list can be omitted from a built-in function reference that has no arguments. In contrast, the parentheses must be given with a programmed function reference even if there are no arguments.

5. Finally, a built-in function reference is evaluated at a relatively low cost. In contrast, the evaluation of a programmed function reference is relatively expensive, even when the invoked procedure consists of only a few simple statements.

This list shows that the differences between built-in and programmed function references are more important than the similarities.

OPERATOR EXPRESSIONS

Like a built-in function reference, an operator expression performs a specific calculation on its arguments and delivers the result as the value of the expression. For each operator, the calculation performed is part of the definition of PL/I. The only difference between operators and built-in function references is a difference in form. An operator expression can be thought of as a built-in operation that, because of its frequency of use, is represented by means of a special notation, using an operator, rather than by means of the less compact built-in function reference.

A specific definition of each of the operators is given in the next section, "Operations". Just as for built-in functions, each definition of an operator gives restrictions on the arguments and gives rules for calculating the value of the result. A quick way to find the definition of a particular operator is to look the operator up in the index of this manual.

The Form of Operator Expressions

An operator expression has one of the following forms:

(arg1 inop arg2)	arg1 inop arg2
(preop arg1)	preop arg1

where arg1 and arg2 are the operands (also called arguments) of the operator, inop is the infix operator, and preop is the prefix operator. An infix operator must be one of the following:

```
**  *  /  +  -  !!
=  ^=  <  ^<  >  ^>  <=  >=
&  !
```

A prefix operator must be one of the following:

```
-  +  ^
```

For both the infix operator and prefix operator a parenthesized and an unparenthesized form is given. It is always correct to use the parenthesized form of an operator expression. The unparenthesized form can be used when the priority rules, described later in this discussion of operator expressions, provide the interpretation that the programmer wants.

The Interpretation of Operator Expressions

An operator expression is interpreted in the same two steps that apply to a built-in function reference, as follows:

1. Storage Type Determination. Determine the target storage type for each operand and the storage type of the result. The rules for determining these storage types are given in the individual definitions of the operators; often they depend on the storage types of the operands.

2. Reference Evaluation. Evaluate each operand and convert it to the target storage type for the operand. Evaluate the reference as specified in the definition of the operator.

Step 1 of this interpretation is performed by the compiler before the program is executed. Step 2 is performed each time the operator expression is evaluated during program execution.

The Operator Priority Rules

Two operators are on the same expression level if they appear in the same expression and the only parentheses that appear between them are matched pairs. When there are several operators on the same expression level, the following table determines the order in which the operators are evaluated:

Priority	Operators	Order within Priority
highest	** ^ prefix + prefix -	right to left
	* /	
	infix + infix -	
	!!	
	= ^= < ^< > ^> <= >=	left to right
	&	
lowest	!	

When two operators appear on the same expression level, the operator with higher priority is evaluated first. If the operators have the same priority, then they are evaluated in left to right or in right to left order, depending on the entry in the third column of the table. These are the operator priority rules.

As an example of the application of the operator priority rules, consider the following expression:

```
4*(A-(B/C)**2+D)
```

There are three expression levels in this expression. One of them has just one operator, '*', and another also has just one operator, '/'. However, the remaining expression level contains three operators, '-', '**', and '+'. The rules are applied as follows:

- The operator with the highest priority is '**'; therefore, this operator is evaluated before the other operators on the same expression level.
- The remaining two operators, '-' and '+', both have the same priority. According to the table, they are evaluated from left to right; therefore, '-' is evaluated before '+'.

According to this analysis, the example is equivalent to the following expression:

```
4*((A-((B/C)**2))+D)
```

This expression makes the required order of evaluation explicit.

In some cases, the results of the operator priority rules are consistent with well-known conventions of mathematical notation. In other cases, the results do not correspond in any obvious way to familiar notation. Some examples of the latter are:

Given	Equivalent
A/B/C	(A/B)/C
A**B**C	A**(B**C)
-A**B	-(A**B)
A*-B	A*(-B)
A=B=C	(A=B)=C

In each of these cases it is suggested that the given expression be avoided and that, instead, the parenthesized equivalent be used. Parentheses should not be omitted except where the interpretation will be obvious to everyone who must read the program.

The operator priority rules do not fully determine the order in which operators in an expression are evaluated; it applies only to operators that are on the same expression level. Consider, for example:

```
(A+B)*(C+D)
```

In this expression there are three expression levels, and each contains only one operator; therefore, the operator priority rules say nothing about this example. From the fact that the '+' operators are contained in the operands of the '*', it can be concluded that the '+' operators are evaluated first. However, the order in which the two '+' operators are evaluated is not defined.

Examples of Operator Expressions

As an example of the use of operator expressions, consider the following program fragment:

```
DCL (Y,A,B,C,D) FLOAT;
...
Y = A*(B-C)+D;
```

According to the priority rules, the right-hand side of the assignment statement is equivalent to:

```
(A*(B-C))+D
```

The right-hand side is made up of three operator expressions.

According to the definition of the '*', '-', and '+' operators, the target storage type for all these operands is:

```
REAL FLOAT BINARY(27)
```

When the definitions of the operators are applied to this expression, it turns out that the target storage type for every operand and the result storage type for every operator is also:

```
REAL FLOAT BINARY(27)
```

This simplicity with respect to storage types is typical of scalar, floating-point calculations.

As a more complicated example of the case of operator expressions, consider:

```
DCL I FIXED;
...
I = I+1;
```

In this example, the assignment statement increases the value of I by one. The calculation is simple, but the determination of the storage type of the right-hand-side expressions is not.

The following table gives the operand storage types and the target storage types as determined by the definition of the '+' operator.

Operand	Operand Storage Type	Target Storage Type
I	REAL FIXED BINARY(17)	REAL FIXED BINARY(17)
1	REAL FIXED DECIMAL(1)	REAL FIXED BINARY(4)

The storage type of the result is:

```
REAL FIXED BINARY(18)
```

Observe that the precision of the target and result storage type accommodates any value of X that can arise.

SECTION IX

OPERATIONS

The operations of PL/I are invoked by the operators and the built-in functions. It is the operations that determine the computational foundation of PL/I. There are well over a hundred operations, and they are described here under the following headings:

- Arithmetic Operations. These operations perform the fundamental operations of arithmetic. They range from such elementary operations as addition and subtraction to less well known operations such as the modulus and the conjugate.

- Mathematical Operations. These are the transcendental functions of applied mathematics, including the exponential and logarithmic functions and the standard trigonometric functions.

- String Operations. These operations manipulate string values. They range from the fundamental concatenate operator and SUBSTRING function to advanced functions for text processing.

- Address and Area Operations. These functions manipulate address values and area values. They are rarely used outside of advanced programming applications.

- Array Operations. These operations are especially designed to operate on array arguments.

- Conversion Operations. These operations are used to convert a value of one storage type to another.

- Special Operations. These operations are intimately related to the PL/I processor; they access system variables or depend on the details of program execution.

The definitions given here show the result of a given operation applied to given argument values. The definitions do not discuss the context of the operation. The way in which operations fit into the context of a program is described in the preceding section on "Expressions".

Following this introduction, this section gives general rules and conventions that apply to all of the definitions of operations. The section then continues through seven subsections, each of which begins by giving the conventions that apply to that subsection. In many cases, a complete understanding of a given operation requires familiarity with both the general conventions and the conventions for the subsection in which the given operation is defined.

GENERAL REMARKS

Remarks that apply to all of the definitions in this section are given here. They apply to argument evaluation, nonstandard operations, and the conventions for the presentation of examples.

Argument Evaluation

The evaluation of the arguments of an operator expression or a built-in function reference is described in detail in the preceding section, "Expressions". Three important points are stated briefly here:

- When an operator expression or a built-in function reference has several arguments, the order in which the arguments are evaluated is not defined. The processor may, for example, begin the evaluation of the first argument, perform the evaluation of the second argument, and then complete the evaluation of the first argument.

- Except where the definition of an operation includes a restriction to the contrary, one or more of the arguments can be an aggregate. If the arguments of the operation have different aggregate types, then at least one of the arguments must have an aggregate type that is a suitable target for the conversion of the other arguments; before the operation is performed, all the arguments are converted to this aggregate type according to the rules given earlier, in the section on "Value Conversion". When the arguments all have the same aggregate type (either by conversion or because they were given that way), the operation is applied to corresponding scalar components of the arguments just as it would be applied to scalar arguments. The result has the same aggregate type as the arguments. Examples of operations on aggregate arguments are given earlier, in the section on "Expressions".

- Standard PL/I allows implicit conversion between any computational data types. However, GCOS PL/I, as a matter of policy, discourages implicit conversion between the following general types of values:

  arithmetic
  character string
  bit string

  The GCOS PL/I compiler prints a warning message whenever implicit conversion between these general types is required by context. Therefore, the definitions given in this section specify that a given argument must be one of these three major types. When it is necessary, for example, to use a character-string argument for an operation that requires an arithmetic value, one of the functions described under "Conversion Operations" in this section should be used to perform the conversion explicitly.

Nonstandard Operations

The GCOS implementation of PL/I includes certain operations that are not part of Standard PL/I. The heading of the definition of each such operation ends with an asterisk(*). There are two kinds of nonstandard functions. The first kind is an extension of PL/I and is designed to make programming easier; the "Hyperbolic Functions" are examples. This kind of operation can be eliminated quite easily in the event a program must be reduced to Standard PL/I. The second kind of nonstandard function operation depends on the way GCOS PL/I is implemented; the "Implementation-Dependent Address Functions" are examples. A program that uses these operations must be substantially revised to reduce it to the Standard.

Conventions for Examples

Many examples are given in this section. In each example, the values of the arguments and the result are given as constants whose data types are correct for the example. Consider the following example of the plus-sign operator:

```
(006.) + (-2.00) = 0004.00
```

This example not only shows that the value of six plus minus two is four, but also shows that a DEC(3) value plus a DEC(3,2) value yields a DEC(6,2) result. Often the value of the result is obvious and the purpose of the example is to show the storage type of the result.

Each example consists of an operator expression or a function reference, followed by an equals sign, followed by a result. (The example given in the previous paragraph has this form.) The equals sign means "yields the result"; it is not used as a PL/I symbol.

For certain argument values, the evaluation of an operator expression or a function reference causes a condition to occur; in that case, the condition name, enclosed in parentheses, is given instead of the result. An example is:

```
(2.0000E0) / (0.0000E0) = (ZERODIVIDE)
```

This example shows that division by zero causes the ZERODIVIDE condition to occur.

In many of the examples that have arithmetic arguments, DECIMAL rather than BINARY arguments are shown. The DECIMAL base is used for these examples because it is easier to understand. The use of DECIMAL base is not an endorsement; in fact, DECIMAL values are rarely used in practice outside of business programming.

THE ARITHMETIC OPERATIONS

The arithmetic operations manipulate arithmetic values in a fundamental and relatively general way. A single operation, such as the '+' operator, can have many interpretations depending on the storage types of its arguments. The arithmetic operations contrast with the mathematical operations, described later, which perform transcendental operations and always produce a floating-point result.

The rules for determining the data type of the result of an arithmetic operation are sometimes complicated. Much of the complexity arises in determining the scale-factor of a FIXED result. In many programming applications, it is possible to avoid FIXED data that has a nonzero scale factor; that is, fixed-point data can be confined to integers. In these applications, the rules become simpler.

Conventions for Definitions

The arithmetic operations defined here have some points in common, and certain conventions and general rules apply to all of the definitions.

ARGUMENTS

As the first step in the interpretation of an operation, the storage types are checked. Each arithmetic operation requires arguments that yield arithmetic values; in some cases, an argument is further required to be either REAL or COMPLEX. These requirements are indicated by the use of the following conventions:

X, X1, X2 ... The value of the argument must be arithmetic

R, R1, R2 ... The value of the argument must be REAL arithmetic

Z, Z1, Z2 ... The value of the argument must be COMPLEX arithmetic

For example, the definition of the TRUNC function requires that it have the form:

```
TRUNC(R)
```

Therefore, the use of an argument that does not yield a REAL arithmetic value is invalid.

RESULTS

Each definition gives the precision attribute of the result without regard for the capacity of a particular implementation of PL/I. For each implementation of PL/I, the defined precision must be limited to the capacity of the host computer. The defined precision is modified to a GCOS precision as follows:

Defined	GCOS
FIXED BIN(p,q)	FIXED BIN(MIN(MAX(p,1),71),MIN(MAX(q,-128),127))
FLOAT BIN(p)	FLOAT BIN(MIN(MAX(p,1),63))
FIXED DEC(p,q)	FIXED DEC(MIN(MAX(p,1),59),MIN(MAX(q,-128),127))
FLOAT DEC(p)	FLOAT DEC(MIN(MAX(p,1),59))

These rules simply require that the precision of the result of an operation lies within the ranges for the number-of-digits and scale-factor that were given earlier, in the section on "Value Storage".

THE COMMON DATA ATTRIBUTES

Many of the definitions that follow refer to the common data attributes of the arguments. These attributes are as follows:

- The mode attribute is COMPLEX if the modes of the arguments differ; otherwise, it is the common mode of all the arguments.
- The scale attribute is FLOAT if the scales of the arguments differ; otherwise, it is the common scale of all the arguments.
- The base attribute is BINARY if the bases of the arguments differ; otherwise, it is the common base of all the arguments.

For example, suppose that an operation has two arguments with the following data types:

```
REAL FIXED DEC(10)

REAL FLOAT BIN(27)
```

Then the common data attributes are:

```
REAL FLOAT BIN
```

Elementary Operations

There are five operators and four functions that are used to perform the elementary operations of arithmetic. They are:

- The plus and minus operators. Each can be used either as a prefix operator to supply the sign of an expression or as an infix operator to add or subtract one expression from another.

- The multiplication, division, and exponentiation operators. The exponentiation operator accepts non-integer exponents.

- The ADD, SUBTRACT, MULTIPLY, and DIVIDE functions, which are used in fixed-point calculations when it is necessary to specify the precision of the result.

These functions are of general interest.

THE PREFIX SIGN OPERATORS

Parenthesized operator expressions for the prefix sign operators have the forms:

```
( + X )
( - X )
```

The result is the unchanged value or the negated value, respectively, of X. The storage type of the result is the same as that of X. Examples are:

```
+ (+0435.241) = +0435.241
- (+0435.241) = -0435.241

+ (-2.8192E0) = -2.8192E0
- (-2.8192E0) = +2.8192E0

+ (+1.3000E0-17.891E01) = +1.3000E0-17.891E01
- (+1.3000E0-17.891E01) = -1.3000E0+17.891E01
```

THE INFIX SIGN OPERATORS

Parenthesized operator expressions for the infix sign operators have the forms:

```
( X1 + X2 )
( X1 - X2 )
```

The result is the sum or difference, respectively, of the operands. Before the operation is performed, the operands are converted to the common data attributes, defined earlier. The result has the same data type as the converted operands except:

- If the converted operands are FLOAT with precisions (p1) and (p2), then the result has the precision:

  (MAX(p1,p2))

- If the converted operands are FIXED with precisions (p1,q1) and (p2,q2), then the result has the precision:

  (MAX(p1-q1,p2-q2)+MAX(q1,q2)+1, MAX(q1,q2))

  Observe that a FIXED result has a digit position for each digit position in either operand and, further, an extra, high-order digit position.

Examples are:

```
(2.000E0) + (2.0E0)     = 4.000E0
(2.000E0) + (2)         = 4.000E0
(2.000E0) + (00010.0B) = 100.00000000000E0B
(2.000E0) + (2+0I)      = 4.000E0+0.000E0I

(5555) + (.333)    = 05555.333
(5555) + (.ØØØ23) = 05555.00023
```

In the last example, each slashed zero is a filler zero; therefore, the second argument has precision (2,5), not (5,5).

THE MULTIPLICATION OPERATOR

A parenthesized operator expression for the multiplication operator has the form:

```
( X1 * X2 )
```

The result is the product of the operands. Before the operation is performed, the operands are converted to the common data attributes defined earlier. The result has the same data type as the converted operands except:

- If the converted operands are FLOAT with precisions (p1) and (p2), then the result has the precision:

  ```
  ( MAX(p1,p2) )
  ```

- If the converted operands are FIXED with precisions (p1,q1) and (p2,q2), then the result has the precision:

  ```
  ( p1+p2+1, q1+q2 )
  ```

  Observe that a FIXED result has more digits than either operand; thus multiplications tend to increase the precision of an expression.

Examples are:

```
(2.000E0) * (-3.0E0) = -6.000E0
(002.00) * (-003.00) = -0000006.0000
(002.00) * (-003.00) * (004.00) = -00000000024.000000
```

THE DIVISION OPERATOR

A parenthesized operator expression for the division operator has the form:

(X1 / X2)

The result is the quotient of X1 by X2. If X2 is zero, then the ZERODIVIDE condition occurs, and any attempt to resume execution at the division operation is invalid. Before the operation is performed, the operands are converted to the common data attributes defined earlier. The result has the same data type as the converted operand except:

- If the converted operands are FLOAT with precision (p1) and (p2), then the result has the precision:

  (MAX(p1,p2))

- If the converted operands are FIXED with precisions (p1,q1) and (p2,q2), respectively, then the result has the precision:

  (N, N-p1+q1-q2)

  where N is the maximum number-of-digits, 59 for DECIMAL and 71 for BINARY. Observe that a FIXED result always is of maximum size.

The precision of a FIXED result is of maximum size because PL/I cannot make a better choice on the basis of available information. In virtually all cases, the programmer will have better information; for fixed-point division, he should use the DIVIDE function, which allows specification of the correct result precision. Examples of the division operator are:

```
(6.000E0) / (-3.0E0) = -2.000E0

1/3     = 0.333...3     (59 digits)

2+1/3   = 02.333...3 (60 digits) = 2.333...3 (59 digits)
22+1/3  = 22.333...3 (60 digits) = (SIZE)
```

The fact that the small and useful expression '22+1/3' cannot be used without producing an occurrence of the SIZE condition shows the weakness of the fixed-point divide operation. The expression could be written as:

```
22+DIVIDE(1,3,10,10) = 022.3333333333
```

THE EXPONENTIATION OPERATOR

A parenthesized operator expression for the exponentiation operator has the form:

```
(X1 ** X2)
```

The result is X1 to the power X2. For some values, the mathematical value of the operator is indeterminate; for these values the PL/I operator is defined as follows:

- When X1 is zero, the function is defined as follows:

  ```
  if X2 is REAL and X2>0 or
     X2 is COMPLEX and REAL(X2)>0 and IMAG(X2)=0

  then 0**X2 = 0;

  otherwise, the ERROR condition occurs.
  ```

- When X2 is zero, the function is defined as follows:

  ```
  if X1^ = 0

  then X1**0 = 1;

  otherwise, the ERROR condition occurs.
  ```

- When X1 is REAL and negative, the function is defined as follows:

  ```
  if X2 is a decimal integer constant (without a sign),

  then (-X1)**X2 = ((-1)**X2)*(X1**X2);

  otherwise, the ERROR condition occurs.
  ```

  This restriction excludes a complex value for a REAL result; but observe that if X2 is supplied in any way other than a constant, the ERROR condition occurs (even if its value is a positive integer).

Before the operation is performed, the operands are converted to the common data attributes defined earlier; however, for exponentiation the following exception applies:

```
If the second argument, the exponent, has the original data type:

    REAL FIXED base(p2,0)

then the adjusted data type for the second operand retains the
attributes REAL and FIXED regardless of the attributes of the first
operand.
```

This exception is perceptible only when a conversion error can occur; otherwise, it can be ignored.

THE ADD, SUBTRACT, MULTIPLY, AND DIVIDE FUNCTIONS

References to these functions have the forms:

```
ADD(X1,X2,P)          ADD(X1,X2,P,Q)
SUBTRACT(X1,X2,P)     SUBTRACT(X1,X2,P,Q)
MULTIPLY(X1,X2,P)     MULTIPLY(X1,X2,P,Q)
DIVIDE(X1,X2,P)       DIVIDE(X1,X2,P,Q)
```

where P and Q cannot be general expressions but must instead be given as decimal integer constants (Q can be signed). These functions are equivalent to the corresponding operators in all respects but one:

> The precision of the result of each of these functions is given explicitly by P (number of digits) and Q (scaling factor).

If the result is FIXED and Q is not given, then Q=0 is assumed. If the result is FLOAT and Q <u>is</u> given, then the reference is invalid.

Comparison Operations

There are eight operators and two functions that are used to compare arithmetic values. They are:

- The relational operators, including '=', '<', and six similar operators. Only the application of these operators to arithmetic values is described here.
- The MIN and MAX functions, which yield the smallest or largest member, respectively, of a given list of arithmetic values.

These functions are of general interest.

THE RELATIONAL OPERATORS FOR ARITHMETIC VALUES

A parenthesized expression for a relational operator has the form:

(X1 <u>op</u> X2)

where the operator, <u>op</u>, is one of the following:

```
=       ( is equal to )
<       ( is less than )
>       ( is greater than )
<=      ( is less than or equal to )
>=      ( is greater than or equal to )
^=      ( is not equal to )
^<      ( is not less than )
^>      ( is not greater than )
```

The arguments must be REAL unless the operator is '=' or '^='. The relational operators can apply to many types of values, as follows:

- If either operand yields an arithmetic value or a character string declared with a numeric picture, then the expression is an arithmetic comparison and is described here.

- If both operands yield string values and neither is declared with a numeric picture, then the expression is a string comparison and is described later in this section under "The Relational Operators for String Values".

- If both operands yield address values, then the expression is an address comparison and is described later in this section under "The Relational Operators for Address Values".

The result of a comparison expression is "1"B or "0"B depending on whether the comparison is true or false. The arguments are converted to the common data attributes. The result has storage type BIT(1) NONVARYING. Examples are:

```
+015.03 =  +15.0300  = "1"B
+015.03 <  +15.0300  = "0"B
+015.03 <= +15.0300  = "1"B

+15.030E0 =  +15.03  = "1"B
+15.030E0 <  +15.031 = "1"B
+15.030E0 ^< +15.031 = "0"B
```

THE MIN AND MAX FUNCTIONS

References to these functions have the forms:

```
MIN(R1,R2,...)
MAX(R1,R2,...)
```

where the arguments must be REAL and where each function can have any number of arguments, provided it has more than one argument. The result is the minimum value or the maximum value, respectively, in the set of values given by the arguments. The arguments must yield REAL values; these values are converted to the common data attributes defined earlier in this section. The result has the storage type of the converted arguments except:

- If the converted arguments, R1, R2, and so on, are FLOAT and have precisions (p1), (p2), and so on, then the precision of the result is:

  (MAX(p1,p2,...))

- If the converted arguments, R1, R2, and so on, are FIXED and have precisions (p1,q1), (p2,q2), and so on, the precision of the result is:

  (MAX(p1-q1,p2-q2,...)+MAX(q1,q2,....),MAX(q1,q2,....))

These expressions provide the precision that is necessary to accommodate any argument without loss of digits. Examples are:

```
MIN(2.000E0,-000.5,11.1000000E0) = -5.00000000E-1
MAX(-3,99,99.05,00.013) = 99.050
```

Truncating Functions

There are five functions that change a value by discarding digits from a representation of the value. They are:

- The TRUNC, FLOOR, and CEIL functions, which use three different rules for discarding all fractional digits.
- The ROUND function, which discards any sequence of low order digits by rounding.
- The MOD function, which, in one form of usage, discards any sequence of high order digits.

These functions are of general interest.

THE TRUNC, FLOOR, AND CEIL FUNCTIONS

References to these functions have the forms:

```
TRUNC(R)
FLOOR(R)
CEIL(R)
```

where the argument must be REAL. The results of these functions are the truncation, the floor, and the ceiling, respectively, of R. Each function discards the fractional part of its argument to produce an integer-valued REAL result.

- If R is an integer:

```
TRUNC(R) = R
FLOOR(R) = R
CEIL(R)  = R
```

- If R is not an integer:

```
TRUNC(R)   is the result of discarding the fractional digits of R
FLOOR(R)   is the next integer below R
CEIL(R)    is the next integer above R
```

The result has the storage type of the argument except that, for a FIXED argument, the precision (p,q) becomes

(MAX(p-q+1,1),0)

That is, a FIXED result is an integer with an added high-order digit. Examples of the functions are:

R	TRUNC(R)	FLOOR(R)	CEIL(R)
1.782E0	1.000E0	1.000E0	2.000E0
001	0001	0001	0001
-9.56	-09.	-10.	-09.
-.0003	+0.	-1.	+0.

Observe that FLOOR(-9.56) makes use of the added high-order digit of the result.

THE ROUND FUNCTION

A reference to the function has the form:

ROUND(X,Q)

where Q cannot be a general expression but must instead be given as an optionally signed decimal integer. The result has the same mode as X and is a rounding of the value of X. Rounding is defined as follows:

- When a value is rounded to n digits, the digits after the nth digit are dropped and the nth digit is increased by one if the (n+1)th digit is 5 or greater (for decimal) or 1 (for binary).

When X is REAL, the result of the function is defined as follows:

- If X is FLOAT, Q must be greater than 0 and the mantissa is rounded to Q digits.

  If X is FIXED, it is rounded to a value that has Q fractional digits.

The result has the same storage type as X except that

- If X is FLOAT, the precision of the result is (Q).

  If X is FIXED with precision (p,q), the precision of the result is

  (MAX(1,p-q+1+Q),Q)

  Thus a high-order digit is added in case the rounding propagates to a new order of magnitude.

Examples for floating-point are:

```
ROUND(-183.629E6,4)       = -183.6E6
ROUND(-183.629E6,5)       = -183.63E6
ROUND(-183.629E6,6)       = -183.629E6
ROUND(-111010.111E3B,7) = -111011.0E3B
```

Observe that neither the exponent of the value nor the point in the mantissa affect the rounding.

Examples for fixed-point are:

```
ROUND(-183.629,2)   = -0183.63
ROUND(-183.629,-1) = -018Ø. (FIXED DEC(3,-1))
ROUND(-183.629,-5) = +ØØØØØ. (FIXED DEC(1,-5))
```

In these examples, a slashed zero is a filler zero. In the last example, the formula for the precision gives:

```
(p-q+1+Q,Q) = (6-3+1-5,-5) = (-1,-5)
```

However, the number-of-digits must be greater than zero, so the precision is (1,-5). Observe that the position of the point <u>does</u> affect the rounding of fixed values.

For COMPLEX values, the function is defined by

```
ROUND(R1+R2*1I,Q) = ROUND(R1,Q) + ROUND(R2,Q)*1I
```

For example,

```
ROUND(21.56+06.21I,0) = 022.+006.I
```

THE MOD FUNCTION

A reference to the function has the form:

```
MOD(R1,R2)
```

where the arguments must be REAL. The result is REAL and is R1 <u>modulus</u> R2: that is,

- If R2 ^= 0, THEN MOD(R1,R2) = R1 - R2*FLOOR(R1/R2)
- If R2 = 0, THEN MOD(R1,R2) = R1

The arguments must yield REAL values; these values are converted to the common data attributes, defined earlier. The result has the storage type of the converted arguments, except:

- If the converted arguments are FLOAT with precisions (p1) and (p2), respectively, then the result has the precision:

  (MAX(p1,p2))

- If the converted arguments are FIXED and have precisions (p1,q1) and (p2,q2), respectively, then the result has precision:

  (p2-q2+MAX(q1,q2),MAX(q1,q2))

  Thus the result has as many integer digits as R2 and enough fractional digits for either R1 or R2.

When both R1 and R2 are positive, the modulus is the remainder of the conventional integer division of R1 by R2. For example,

```
MOD(42,5) = 2

MOD(129.2867,25.0) = 04.2867

MOD(129.2867E0,25.0E0) = 004.2867E0

MOD(182.00E0,3) = 002.00E0
```

The storage types for these examples are

Converted Arguments	Result
FIXED DEC(2,0), FIXED DEC(1,0)	FIXED DEC(1,0)
FIXED DEC(7,4), FIXED DEC(3,1)	FIXED DEC(6,4)
FLOAT DEC(7), FLOAT DEC(3)	FLOAT DEC(7)
FLOAT DEC(5), FLOAT DEC(5)	FLOAT DEC(5)

When R1 or R2 is negative, the notion of a remainder is not well-defined, so the behavior of the function is less obvious. Consider the four possible combinations of signs:

```
MOD(42,5)   =  2      (FLOOR(42/5 = 8))
MOD(42,-5)  = -3      (FLOOR(42/-5 = -9))
MOD(-42,5)  =  3      (FLOOR(-42/5 = -9))
MOD(-42,-5) = -2      (FLOOR(-42/-5 = 8))
```

When R1 is positive and R2 = 10**n, the function discards digits from the left end of a DECIMAL value until R1<R2. For example:

```
MOD(231.34,100) = 031.34
MOD(231.34,1)   = 0.34
MOD(231.34,.1)  = .04
```

Similarly, for BINARY values:

```
MOD(111110.110B,1000B) = 0110.110B
MOD(111110.110B,.1B)   = .010B
```

Observe that the use of the MOD function for truncation does not work when R1 is negative. That is,

```
MOD(-231.34,1) = 0.66
```

Sign-Manipulation Functions

There are two functions that manipulate the sign of an arithmetic variable. They are

- The ABS function, which sets the sign of a real number to plus and forms the modulus of a complex number.
- The SIGN function, which yields a value that is +1 or -1 depending on the sign of the argument.

These functions are of general interest.

THE ABS FUNCTION

A reference to the function has the form:

```
ABS(X)
```

The result is the absolute value of X. For a REAL argument, the result has the same storage type as the converted argument. Examples are:

```
ABS(-13.284)    = +13.284
ABS(+0019.8)    = +0019.8
ABS(-3.0000E2) = +3.0000E2
ABS(-63.000)    = +63.000
```

For COMPLEX values, the function is defined by:

```
ABS(R1+R2*1I) = SQRT(R1**2 + R2**2)
```

where SQRT is the square root function defined later in this section. The result has the same data type as the COMPLEX argument except that the mode is REAL and, for a FIXED argument, the precision (p,q) becomes (p+1,q).

Consider the function reference:

```
ABS(+8.00+9.00I) = +12.04
```

Observe that the result value in this example makes use of the added digit in the result. Other examples are:

```
ABS(-003.00+003.00I)      = +0004.24
ABS(+20.200E0+.80000E0I) = +20.216E0
ABS(-63.000+00.000I)      = +063.000
```

Compare the last example to the last example for REAL arguments: the values are the same but the number-of-digits is different.

THE SIGN FUNCTION

A reference to the function has the form:

```
SIGN(R)
```

where the argument must be REAL. The result is +1, 0, or -1 depending on whether R is positive, zero, or negative. The result is a 17-bit integer. Examples are:

```
SIGN(-019.32)  = -1.  (as a REAL FIXED BIN(17) value)
SIGN(+8.23E2)  = +1.  (as a REAL FIXED BIN(17) value)
SIGN(+0000.0)  = +0.  (as a REAL FIXED BIN(17) value)
```

Observe that, if R is FLOAT:

```
R = SIGN(R)*ABS(R)
```

However, if R is not FLOAT the two sides of this equation agree in value but not in data type.

Complex Arithmetic Functions

There are three functions that are used to manipulate the real and imaginary parts of a COMPLEX value. They are:

- The REAL and IMAG functions, which yield the real parts of a complex value. These functions can also be used as pseudo-variables, as described in the section on "Value Assignment".
- The COMPLEX function, which yields a complex value from given real parts.
- The CONJG function, which yields the conjugate of a given complex value.

These functions are used only in computations in complex arithmetic.

THE REAL AND IMAG FUNCTIONS FOR ARITHMETIC

References to these functions have the form:

```
REAL(Z)
IMAG(Z)
```

where the argument must be COMPLEX. The result is the real part or imaginary part, respectively, of Z. The result has the same storage type as the argument except the result is REAL. For example:

```
REAL(-23041.E-2+00519.E1I)   =  -23041.E-2
IMAG(-23041.E-2+00519.E1I)   =  +00519.E1
```

The use of these functions for conversion is described later in this section, under "Conversion Operations".

The pseudo-variables named REAL and IMAG are described later, in the section on "Value Assignment".

THE COMPLEX FUNCTION

A reference to this function has the form:

```
COMPLEX(R1,R2)
```

where the arguments must be REAL. The result is a complex value whose real part is the value of R1 and whose imaginary part is the value of R2. Before interpretation of the function, R1 and R2 are converted to arithmetic values of the common data attributes defined earlier. The result has the same storage type as the converted arguments except that the result is COMPLEX. For example:

```
COMPLEX(-23041.E-2,+0051.E1) = -23041.E-2+00519.E1I
COMPLEX(-23041E-2,0)         = -23041.E-2+00000.E0I

COMPLEX(0,+00519.E1) = +00000.E0+00519.E1I

COMPLEX(16,"1011"B)      = 0...010000B+0...01011BI (71 digits each)
COMPLEX(16,"23041.E-2") = 0...016+0...230I (59 digits each)
```

THE CONJG FUNCTION

A reference to the function has the form:

```
CONJG(Z)
```

where the argument must be COMPLEX. The result is the conjugate of Z, that is:

```
CONJG(R1+R2*1I) = R1-R2*1I
```

The result has the same storage type as the argument. For example:

```
CONJG(-05.000+02.0001)    = -05.000-02.0001
CONJG(+2.312E0-.4231E01) = +2.312E0+.4231E01
CONJG(+0002+00001)        = +0002+00001
```

MATHEMATICAL OPERATIONS

The functions described here always produce floating-point results. These functions contrast with the arithmetic functions, which produce fixed-point results for some arguments and floating-point results for others.

These functions are called "mathematical" because they are usually required only in mathematical, scientific, and engineering applications of PL/I. They are all transcendental functions and cannot, therefore, be defined in terms of a single expression using the basic arithmetic operations.

Conventions for Definitions

The functions defined here have many points in common. The data types of arguments and results are handled in a uniform way, and the same methods are applied to the calculation of the result value. Therefore, the following general conventions can be applied to all of the definitions.

ARGUMENTS

All of the functions operate on FLOAT arguments. If an argument is FIXED, then its value is converted to a FLOAT target whose number-of-digits is the same as that of the given argument. For example:

Argument	Target for Conversion
REAL FIXED BIN(30,5)	REAL FLOAT BIN(30)
COMPLEX FIXED DEC(10,0)	COMPLEX FLOAT DEC(10)
COMPLEX FIXED BIN(70,0)	COMPLEX FLOAT BIN(63)

The last example shows that the number-of-digits is not allowed to exceed the maximums for FLOAT values (63 for BINARY and 59 for DECIMAL).

For a two-argument function (ATAN and ATAND), the arguments are converted to a FLOAT target with a common base. If the arguments have different bases, then the target base is BINARY.

Some of the mathematical built-in functions require REAL arguments. The following symbols are used in the definitions:

R, R1, R2, ...	The mode of the operand must be REAL.
X, X1, X2, ...	The mode of the operand is not restricted.

If a COMPLEX argument is used where a REAL argument is required, the function reference is invalid. An implicit conversion from COMPLEX to REAL is not assumed.

RESULTS

The storage type of the result of a mathematical function is the same as the storage type of its converted argument(s).

EVALUATION

The GCOS implementation of the mathematical functions uses binary arithmetic. If the argument is DECIMAL(p), then it is converted to BINARY(63). When the result of the function is obtained, it is converted back to DECIMAL(p). As a result of this policy, a decimal result cannot be accurate to more than about 20 decimal digits.

Some functions are not defined for certain arguments. In such cases, the definition of the function says that the ERROR condition occurs. The ERROR condition is used for any error that does not have its own, more specialized, condition. When the ERROR condition occurs, the execution of the program cannot resume at the point of interruption, so some disruption is inevitable. The details are given in the section on "Condition Handling".

A function can have a COMPLEX result if and only if its argument is COMPLEX. Thus a REAL argument that would produce a COMPLEX result is an error. Consider SQRT(X). If X is REAL and X<0, then the ERROR condition occurs; but if X is COMPLEX, the ERROR condition does not occur.

Some functions are potentially multi-valued. For such a function, there is more than one result value that satisfies the mathematical definition of the function. In each such case, PL/I imposes conditions on the result that provide a unique value. For example, in the definition of SQRT, the following condition is given:

For X REAL: SQRT(X) >= 0

Thus the negative square root is excluded and the result is uniquely defined.

The mathematical definitions of the functions as they apply to real arguments are not given; it is assumed that the reader who uses these functions is familiar with their definition. On the other hand, the definitions of the functions as they apply to complex arguments are given. These definitions are given because, although they are part of standard mathematics, they are not easy to find. The definitions are expressed in terms of operations on REAL values and make use of valid PL/I expressions.

In some of the definitions, the mathematical constants 3.14159... and 2.71828... are referred to by the names pi and e, respectively. These names are not built into the PL/I language. If a programmer needs these names, he must declare and set them just as he would any other variable.

Functions Related to Exponentiation

There are five built-in functions that relate to the exponentiation operation. They are:

- The EXP function, which raises the mathematical constant e = 2.71828... to a given power.
- The LOG function, which is the inverse of the EXP function.
- The LOG10 and LOG2 functions, which are the inverses of 10**R and 2**R, respectively.
- The SQRT function, which raises a given number to the 1/2 power.

These functions are fundamental to scientific and engineering applications.

THE EXP FUNCTION

A reference to the function has the form:

```
EXP(X)
```

The result is e raised to the power X, where e is the base of the system of natural logarithms. Examples are:

```
EXP(1.0000E0)   = 2.7183E0
EXP(0.0000E0)   = 1.0000E0
EXP(-1.0000E0) = .36788E0
```

For complex values, the function is defined by:

```
EXP(R1+R2*1I) = EXP(R1)*(COS(R2)+SIN(R2)*1I)
```

Thus, for example,

```
EXP(0.0000E0+.78540E0I) = .70711E0+.70711E0I
```

THE LOG FUNCTION

A reference to the function has the form:

```
LOG(X)
```

The result is the logarithm base-e, or natural logarithm, of X. The function is the inverse of the EXP function. Examples are:

```
LOG(2.7183E0) =  1.0000E0
LOG(1.0000E0) =  0.0000E0
LOG(.36788E0) = -1.0000E0
```

The argument is restricted as follows:

- If X is REAL, then X<0 causes the ERROR condition to occur because the result is a complex value.
- If X is REAL or COMPLEX, X=0 causes the ERROR condition to occur because the result is not defined.

The result satisfies the conditions:

For X REAL:	The requirement that the result must be REAL is sufficient to make the result unique.
For X COMPLEX:	-pi < IMAG(LOG(X)) <= pi

For complex values, the function is defined by:

```
LOG(R1+R2*1I) = .5*LOG(R1**2+R2**2) + ATAN(R2,R1)*1I
```

Thus, for example:

```
LOG(.70711E0+.70711E0I) = .45481E-5+.78540E0I
```

(observe that .70711 is approximately 1/SQRT(2) and .45481E-5 is close to zero.

THE LOG10 AND LOG2 FUNCTIONS

References to these functions have the forms:

```
LOG10(R)
LOG2(R)
```

where the argument must be REAL. The results are the logarithm base-10 and logarithm base-2, respectively, of R. Examples are:

```
LOG10(1.0000E4) = 4.0000E0
LOG10(2.0000E4) = 4.3010E0
LOG2(64.000E0)  = 6.0000E0
```

The following restriction applies to the argument:

- For both LOG10 and LOG2, if R<=0, then the error condition occurs because a real result does not exist.

These functions can be used to determine how many digits are required for the representation of a given integer in either decimal or binary representation. For example,

```
CEIL(LOG2(70)) = CEIL(6.1) = 7
```

which shows that 70 requires seven digits for its binary representation.

THE SQRT FUNCTION

A reference to the function has the form:

```
SQRT(X)
```

The result is a square root of X. Examples are:

```
SQRT(0.0000E0) = 0.0000E0
SQRT(1.0000E0) = 1.0000E0
SQRT(2.0000E0) = 1.4142E0
```

The argument is restricted as follows.:

If X is REAL, then X < 0 causes the ERROR condition to occur because the result is a complex value.

The result satisfies the conditions:

```
For X REAL:     SQRT(X) >= 0

For X COMPLEX:  either REAL(SQRT(X)) > 0
                  or REAL(SQRT(X)) = 0 and IMAG(SQRT(X)) >= 0
```

For complex values, the function is defined by:

```
SQRT(R1+R2*1I) = SQRT((R3+R1)/2) + SQRT((R3-R1)/2)*1I
```

where

```
R3 = SQRT(R1**2+R2**2)
```

Thus, for example:

```
SQRT(3.0000E0+4.0000E0I) = 2.0000E0+1.0000E0I
```

Trigonometric Functions

There are ten built-in functions for trigonometry. They are:

- The SIN, COS, and TAN functions, which assume that their arguments are in radians.
- The SIND, COSD, and TAND functions, which assume that their arguments are in degrees.
- The ordinary ATAN and ATAND functions, which each have one argument and are the conventional inverses of the TAN and TAND functions.
- The Cartesian ATAN and ATAND functions, which each have two arguments and have results which range over all four quadrants of the coordinate system.

The functions are not the usual textbook selection; instead, the functions are selected and designed for the needs of practical computing. For example, the secant function is omitted because it is rarely used and can be expressed in terms of the COS built-in function. On the other hand, the Cartesian ATAN function, rarely mentioned in trigonometry texts, is included because it is needed for calculation of complex values and certain other calculations.

THE SIN, COS, AND TAN FUNCTION

References to these functions have the forms:

```
SIN(X)
COS(X)
TAN(X)
```

The result is the sine, cosine, or tangent, respectively, of X. X is assumed to be expressed in radians. Examples are:

```
SIN(0.0000E0) =  0.0000E0
COS(3.1416E0) = -1.0000E0
TAN(.78540E0) =  1.0000E0
```

For complex values, the function is defined by:

```
SIN(R1+R2*1I) = SIN(R1)*COSH(R2) + COS(R1)*SINH(R2)*1I
COS(R1+R2*1I) = COS(R1)*COSH(R2) - SIN(R1)*SINH(R2)*1I
TAN(R1+R2*1I) = SIN(R1+R2*1I)/COS(R1+R2*1I)
```

Thus, for example:

```
SIN(0.0000E0+1.0000E0I) = 0.0000E0+1.1752E0I
COS(0.0000E0+1.0000E0I) = 1.5431E0+0.0000E0I
TAN(0.0000E0+1.0000E0I) = 0.0000E0+.76159E0I
```

THE SIND, COSD, AND TAND FUNCTIONS

References to these functions have the forms:

```
SIND(R)
COSD(R)
TAND(R)
```

where the argument must be REAL. The result is the sine, cosine, or tangent, respectively, of R. R is assumed to be expressed in degrees. For example:

```
SIND(0.0000E0) =  0.0000E0
COSD(180.00E0) = -1.0000E0
TAND(45.000E0) =  1.0000E0
```

THE ORDINARY ATAN FUNCTION

A reference to the function has the form:

```
ATAN(X)
```

The result is the arctangent, expressed in radians, of X. Examples are:

```
ATAN(1.0000E0)  = .78540E0
ATAN(0.0000E0)  = 0.0000E0
ATAN(1.0000E10) = 1.5708E0
```

The argument is restricted as follows:

If X is COMPLEX, then X=+1I or X=-1I causes the ERROR condition to occur because the result is not defined.

The result satisfies the conditions:

```
For X REAL:     -pi/2 < ATAN(X) < pi/2
For X COMPLEX:  -pi/2 < REAL(ATAN(X)) < pi/2
```

For complex values, the function is defined in terms of the ATANH built-in function:

```
ATAN(X) = -1I*ATANH(X*1I)
```

Thus, for example:

```
ATAN(0.0000E0-2.0000E0I) = 1.5708E0-.54931E0I
```

where this result is approximately

```
pi/2 - (1/2)*LOG(3)*1I
```

THE ORDINARY ATAND FUNCTION

A reference to the function has the form:

```
ATAND(R)
```

where the argument must be REAL. The result is the arctangent, expressed in degrees, of R. Examples are:

```
ATAND(1.0000E0)   = 45.000E0
ATAND(0.0000E0)   = 0.0000E0
ATAND(1.0000E10)  = 90.000E0
```

The result satisfies the condition:

```
-90 < ATAND(R) < 90
```

THE CARTESIAN ATAN FUNCTION

A reference to the function has the form:

```
ATAN(R1,R2)
```

The arguments must be REAL. This function can be used to calculate the angular coordinate, in radians, of a point in the X-Y plane. Specifically,

Suppose a point is given with the coordinates:

```
y = R1
x = R2
```

Then the result of the function reference ATAN(R1,R2) is the angle of a line that begins at the origin of the coordinate system and passes through the given point. The angle is expressed in radians, is measured counter-clockwise from the X axis, and is in the range:

```
-pi < ATAN(R1,R2) <= pi
```

Because this function puts the angle in the correct quadrant, it simplifies calculations. Observe, however, that the arguments are written in y,x order; this is contrary to conventional mathematical practice, which usually gives coordinates in x,y order. Some examples of the function are:

```
ATAN(0.0000E0,1.0000E0)    =  0.0000E0
ATAN(1.0000E0,1.0000E0)    =  .78540E0    (= (1/4)*pi)
ATAN(9.0000E2,9.0000E2)    =  .78540E0    (= (1/4)*pi)
ATAN(1.0000E0,0.0000E0)    =  1.5708E0    (= (1/2)*pi)
ATAN(1.0000E0,-1.0000E0)   =  1.3562E0    (= (3/4)*pi)
ATAN(0.0000E0,-1.0000E0)   =  3.1416E0    (= pi)
ATAN(-1.0000E0,-1.0000E0)  = -2.3562E0    (= -(3/4)*pi)
```

The arguments must be REAL; furthermore:

R1 = 0 and R2 = 0 cause the ERROR condition to occur because the function is not defined at the origin of the coordinate system.

The function can be defined in terms of the ordinary ATAN built-in function as follows:

```
                    ATAN(Y/X)          for any Y and X > 0
                    pi/2               for Y > 0 and X = 0
ATAN(Y,X)  =        ATAN(Y/X)+pi       for Y >= 0 and X < 0
                    ATAN(Y/X)-pi       for Y < 0 and X < 0
                    -pi/2              for Y < 0 and X = 0
```

The function can be used to obtain the argument of a complex value; that is:

```
ARG(X) = ATAN(IMAG(X),REAL(X))
```

Observe, again, that the order of arguments is the reverse of the usual order.

THE CARTESIAN ATAND FUNCTION

A reference to the function has the form:

```
ATAND(R1,R2)
```

where the arguments must be REAL. The function is used to compute the angular coordinate, in degrees, of a point in the X-Y plane. The function is defined in terms of the Cartesian ATAN function, as follows:

```
ATAND(R1,R2) = (180/pi)*ATAN(R1,R2)
```

Examples are:

```
ATAND(0.0000E0,1.0000E0)    =  0.0000E0
ATAND(1.0000E0,1.0000E0)    =  45.000E0
ATAND(1.0000E0,-1.0000E0)   =  135.00E0
ATAND(0.0000E0,-1.0000E0)   =  180.00E0
ATAND(-1.0000E0,-1.0000E0)  = -135.00E0
```

The following restriction applies to the argument:

R1 = 0 and R2 = 0 causes the ERROR condition to occur.

Hyperbolic Functions

There are four built-in hyperbolic functions. They are:

- The SINH, COSH, and TANH functions
- The ATANH function, which is the inverse of the TANH function

These functions are used primarily for calculations with complex numbers.

THE SINH, COSH, AND TANH FUNCTIONS*

References to these functions have the forms:

```
SINH(X)
COSH(X)
TANH(X)
```

The result is the hyperbolic sine, hyperbolic cosine, or hyperbolic tangent, respectively, of X. It is assumed that X is expressed in radians. Examples are:

```
SINH(1.0000E0) = 1.1752E0
COSH(1.0000E0) = 1.5431E0
TANH(1.0000E0) = .76159E0
```

For complex values, the function is defined by:

```
SINH(R1+R2*1I) = SINH(R1)*COS(R2) + COSH(R1)*SIN(R2)*1I
COSH(R1+R2*1I) = COSH(R1)*COS(R2) + SINH(R1)*SIN(R2)*1I
TANH(R1+R2*1I) = SINH(R1+R2*1I)/COSH(R1+R2*1I)
```

Thus, for example:

```
SINH(0.0000E0+.78540E0I) = 0.0000E0+.70711E0I
COSH(0.0000E0+.78540E0I) = .70711E0+0.0000E0I
TANH(0.0000E0+.78540E0I) = 0.0000E0+1.0000E0I
```

(where .78540 is an approximation of pi/4.)

THE ATANH FUNCTION*

A reference to this function has the form:

```
ATANH(X)
```

The result is the arc hyperbolic tangent, expressed in radians, of X. Examples are:

```
ATANH(0.0000E0) = 0.0000E0
ATANH(.76159E0) = 1.0000E0
```

The argument is restricted as follows:

If X is REAL, then ABS(X) >= 1 causes the ERROR condition to occur because the result is a complex value.

If X is COMPLEX, then X = +1 or X = -1 causes the ERROR condition to occur because the result is not defined.

The result satisfies the conditions:

For X REAL: (The requirement that the result must be REAL is sufficient to make the result unique.)

For X COMPLEX: -pi/2 < IMAG(ATANH(X)) <= pi/2

For complex values, the function is defined in terms of the LOG built-in function:

```
ATANH(X) = LOG((1+X)/(1-X))/2
```

Functions for Statistical Analysis

There are two built-in functions that relate to statistical calculations. They are:

- The ERF function, which is the error function
- The ERFC function, which is the complement of the error function.

THE ERF AND ERFC FUNCTIONS

References to these functions have the form:

```
ERF(R)
ERFC(R)
```

where the argument must be REAL. The results are the error function and the error function complement, respectively, of R. For example:

```
ERF(0.000E0)  = 0.000E0
ERF(1.000E0)  = .8427E0
ERFC(0.000E0) = 1.000E0
```

The functions are defined by the equations:

$$ERF(R) = (1/SQRT(pi)) * \int_0^R e**(-t**2)dt$$

$$ERFC(R) = 1 - ERF(R)$$

STRING OPERATIONS

The string operations manipulate string-related values. A string-related value is:

a string,
an integer that gives a position in a string, or
an integer that is the length of a string.

Each string operation converts its operands or arguments to string-related values and then computes a result that is also a string-related value.

Conventions for Definitions

The string operations that are defined here have some points in common. In particular, they are uniform in their handling of the storage types, both for arguments and for results. Therefore, the following general conventions can be adopted for use in the definitions.

ARGUMENTS

As the first step in the interpretation of an operation, the storage types of the arguments are checked. In the case of string operations, there are not many possibilities for the storage types, and so the requirements can be indicated by using special symbols for the operands or arguments. For example,

```
( B1 & B2 )
```

means that both operands of the "and" operator should be bit strings.

The complete conventions for the symbols used for the string operations are:

B, B1, B2, ...	The value of the argument should have string-type attribute BIT.
C, C1, C2, ...	The value of the argument should have string-type attribute CHAR.
S, S1, S2, ...	The value of the argument should have string-type attribute BIT or CHAR.
I, J, K, ...	The value of the argument should be arithmetic. It is converted to FIXED BIN(24,0) unless it already has that storage type.

Some remarks on the significance of these conventions are necessary:

- In some cases, two string operands must agree in their lengths. In such cases, blank characters or zero bits are added at the right end of the shorter string until its length is equal to that of the other string.

- Arguments for which the symbol is I, J, K, ... either give a position in a string or the length of a string. The value 24 is used as the number-of-digits for all such arguments because a 24-bit integer is the smallest suitable data type that can accommodate the length of the longest possible bit-string in GCOS PL/I.

Concatenate Operations

The simplest operation on string values is concatenation. For this purpose, PL/I has two built-in operations:

- The concatenate operation, which yields a string that starts with the first operand string and continues with the second.
- The COPY function, which concatenates a given string with itself a given number of times.

These two operations nicely reflect the balance in the set of built-in operations of PL/I. The concatenate operator is fundamental and cannot be expressed in terms of simpler operations. In contrast, the COPY function could easily be programmed by the user in terms of the concatenate operation, but the programmed form would be much less efficient than the built-in COPY function.

THE CONCATENATE OPERATOR

A parenthesized operator expression for the operator has the form:

```
( S1 !! S2 )
```

The result is the concatenation of S1 and S2. For example,

```
( "ABC"!!"DE" ) = "ABCDE"
( "ABCDE"!!"" ) = "ABCDE"
( "ABCDE"!!"ØØ" ) = "ABCDEØØ"
```

If the values of the operands, S1 and S2, are both bit strings, the result is a bit string:

```
( "011"B!!"1101"B ) = "0111101"B
```

THE COPY FUNCTION

A reference to the function has the form:

```
COPY(S,I)
```

The result is the concatenation of I copies of S. For example,

```
COPY("ABC",2) = "ABCABC"
COPY("ABC",0) = ""
```

If S has a bit-string value, the result is a bit-string value:

```
COPY("110"B,2) = "110110"B
COPY("110"B,0) = ""B
```

Substring Operations

There are five built-in operations for obtaining or locating a substring of a given string. They are:

- The SUBSTR function, which yields a substring of the given string that begins at a given position and has a given length.
- The INDEX function, which yields the position in the given string of an occurrence of another given string.
- The BEFORE and AFTER functions, which yield a substring of the given string that occurs before or after an occurrence of a second given string.
- The DECAT function, which is a generalization of the BEFORE and AFTER functions.

The SUBSTR and INDEX functions are fundamental to any string processing. The remaining functions are used primarily in advanced string processing, such as compiling and command-string processing.

THE SUBSTR FUNCTION

A reference to the function has one of the forms:

```
SUBSTR(S,I,J)
SUBSTR(S,I)
```

The result is the substring of S that begins with the Ith character and has length J. For example:

```
SUBSTR("ABCDE",1,4) = "ABCD"
SUBSTR("ABCDE",3,1) = "C"
SUBSTR("ABCDE",3,0) = ""
```

If J is not given, the substring begins with the Ith character and continues to the end of S:

```
SUBSTR("ABCDE",1) = "ABCDE"
SUBSTR("ABCDE",3) = "CDE"
```

Two kinds of errors are possible in the use of SUBSTR, and both cause the STRINGRANGE condition to occur. First, it is an error to use a negative value for J (the length of the substring). Thus:

```
SUBSTR("ABCDE",3,-2) = ? (STRINGRANGE condition)
```

Second, it is an error to attempt to refer to a character that is beyond the end of the given string. Thus:

```
SUBSTR("ABCDE",3,4) = ? (STRINGRANGE condition)
SUBSTR("ABCDE",0,2) = ? (STRINGRANGE condition)
```

There is an exception to this rule: the character after the end of S can be referred to provided it is not used in the substring; this can only happen if the requested string has length 0. Thus:

```
SUBSTR("ABCDE",6,0) = ""
SUBSTR("ABCDE",6) = ""
```

The pseudo-variable named SUBSTR is described later, in the section on "Value Assignment".

THE INDEX FUNCTION

A reference to the function has the form:

```
INDEX(S1,S2)
```

The result is a FIXED BIN(24,0) value that is the position of the beginning of the leftmost occurrence of S2 in S1. For example:

```
INDEX("ABCDE","D") = 4
INDEX("ABCDE","DE") = 4
INDEX("ABCDE","ABCDE") = 1
INDEX("101110"B,"111"B) = 3
```

If S2 is not contained in S1, then the result is zero; thus:

```
INDEX("ABCDE","BXD") = 0
INDEX("","D") = 0
INDEX("101110"B,"010"B) = 0
```

If S2 is the null string, the result is also zero; thus:

```
INDEX("ABCDE","") = 0
```

The function can be used both to locate a substring and to determine whether a substring is present. For some arguments, INDEX is related to SUBSTR as follows:

```
I = INDEX(S,SUBSTR(S,I))    for 0 < I <= LENGTH(S)
```

To illustrate this identity, suppose S is "ABCDE" and I is 2; then,

```
2 = INDEX("ABCDE",SUBSTR("ABCDE",2))
2 = INDEX("ABCDE","BCDE")
2 = 2
```

THE BEFORE FUNCTION

A reference to the function has the form:

```
BEFORE(S1,S2)
```

The result is the substring of S1 that is before the leftmost occurrence of S2 in S1. For example:

```
BEFORE("ABCDE","BC") = "A"
BEFORE("ABCDE","ABC") = ""
BEFORE("ABCDE","E") = "ABCD"
BEFORE("ABCDE","ABCDE") = ""
```

If necessary, a null string is assumed at the beginning of S2; therefore, nothing is before a null string:

```
BEFORE("ABCDE","") = ""
```

If there is no occurrence of S2 in S1, then an occurrence is assumed at the end of S1; therefore, all of S1 is before S2:

```
BEFORE("ABCDE","BXD") = "ABCDE"
BEFORE("","A") = ""
```

If both arguments yield bit strings, then the result is a bit string:

```
BEFORE("011101"B,"11"B) = "0"B
```

THE AFTER FUNCTION

A reference to the function has the form:

```
AFTER(S1,S2)
```

The result is the substring of S1 that is after the leftmost occurrence of S2 in S1. For example:

```
AFTER("ABCDE","BC") = "DE"
AFTER("ABCDE","ABC") = "DE"
AFTER("ABCDE","E") = ""
AFTER("ABCDE","ABCDE") = ""
```

If necessary, a null string is assumed at the beginning of S1 (as with BEFORE); therefore, all of S1 occurs after the null string:

```
AFTER("ABCDE","") = "ABCDE"
```

If there is no occurrence of S2 in S1, an occurrence is assumed at the end of S1 (as with BEFORE); therefore, none of S1 is after S2:

```
AFTER("ABCDE","BXD") = ""
```

If both arguments yield bit strings, then the result is a bit string:

```
AFTER("1110111"B,"0"B) = "111"B
```

The BEFORE and AFTER functions are closely related. The expression

```
BEFORE(S1,S2)!!AFTER(S1,S2)
```

always yields a copy of S1 from which the leftmost substring that matches S2 has been deleted; for example:

```
BEFORE("ABCDE","CD")!!AFTER("ABCDE","CD") = "ABE"
BEFORE("ABCDE","CX")!!AFTER("ABCDE","CX") = "ABCDE"
```

THE DECAT FUNCTION

A reference to the function has the form:

```
DECAT(S1,S2,B)
```

The result is a string that is "deconcatenated" from S1 in a relatively complicated way. The third argument, B, is converted to a bit string of length three. The result string of the DECAT function is thought of as a sequence of three substrings, and each substring is controlled by one of the bits in the value of B, as follows:

- If the first bit is one, then the first substring is BEFORE(S1,S2); otherwise, it is the null string.
- If the second bit is one and S2 is contained in S1, then the second substring is S2; otherwise, it is the null string.
- If the third bit is one, then the third substring is AFTER(S1,S2); otherwise, it is the null string.

Thus, for example, for any S1 and S2:

```
DECAT(S1,S2,"101"B) = BEFORE(S1,S2)!!AFTER(S1,S2)
DECAT(S1,S2,"000"B) = ""
DECAT(S1,S2,"100"B) = BEFORE(S1,S2)
```

If S2 occurs in S1:

```
DECAT(S1,S2,"110"B) = BEFORE(S1,S2)!!S2
DECAT(S1,S2,"111"B) = BEFORE(S1,S2)!!S2!!AFTER(S1,S2)
```

But if S2 does not occur in S1:

```
DECAT(S1,S2,"110"B) = BEFORE(S1,S2)
DECAT(S1,S2,"111"B) = BEFORE(S1,S2)!!AFTER(S1,S2)
```

Some specific examples are:

```
DECAT("ABCDE","D","000"B) = ""
DECAT("ABCDE","D","001"B) = "E"
DECAT("ABCDE","D","010"B) = "D"
DECAT("ABCDE","D","011"B) = "DE"
DECAT("ABCDE","D","100"B) = "ABC"
DECAT("ABCDE","D","101"B) = "ABCE"
DECAT("ABCDE","D","110"B) = "ABCD"
DECAT("ABCDE","D","111"B) = "ABCDE"
```

The primary purpose of the DECAT function is to provide eight functions under a single name. A secondary purpose is to allow convenient selection of a function at execution time; this is done by using a variable expression for B.

Relational, Length, and Reverse Operations

After the concatenation and substring operations have been considered, only a few built-in operations remain that can apply to both character and bit strings. They are:

- The relational operators, which includes '=', '<', and six other similar operators. Only the application of these operators to string values is defined here.
- The LENGTH function, which yields the length of a given string.
- The REVERSE function, which yields a string that contains the characters or bits of the given string but in reverse order.

The relational operators (as applied to strings) are primarily used in advanced string processing, especially in sorting text data. The LENGTH function is fundamental to any string manipulation. The REVERSE function is used in advanced string processing.

THE RELATIONAL OPERATORS FOR STRING VALUES

A parenthesized expression for a relational operator has the form:

(S1 op S2)

where the relational operator, op, is one of the following:

=	(is equal to)
<	(is less than)
>	(is greater than)
<=	(is less than or equal to)
>=	(is greater than or equal to)
^=	(is not equal to)
^<	(is not less than)
^>	(is not greater than)

The relational operators can apply to many kinds of values, as follows:

- If both operands yield string values and neither is declared with a numeric picture attribute, then the expression is a string comparison and is described here.
- If either operand yields an arithmetic value or a character string declared with a numeric picture attribute, then the expression is an algebraic comparison and is described earlier in this section under "The Relational Operators for Arithmetic Values".
- If both operands yield address values, then the expression is an address comparison and is described later in this section under "The Relational Operators for Address Values".

The result of a comparison expression is "1"B or "0"B depending on whether the comparison is true or false. When the operand values have the same length and type, they can be equal only if they are identical. For example,

```
( "ABCDE" = "ABCDE" ) = "1"B
( "ABCDE" = "ABCDX" ) = "0"B
( "ABCDE" ^= "ABCDX" ) = "1"B
( "1101"B = "1101"B ) = "1"B
```

When the operand values have different lengths, characters or bits are added to the right end of the shorter operand value. Blank characters are added to a character string; zero bits are added to a bit string. For example:

```
( "ABCDE" = "ABC" ) = ( "ABCDE" = "ABC␣␣" ) = "0"B
( "11"B = "111"B) = ( "110"B = "111"B ) = "0"B
```

Because of the extension of a short operand value, operand values that are not originally identical can satisfy the '=' operator; for example:

```
( "ABC␣␣" = "ABC" ) = ( "ABC␣␣" = "ABC␣␣" ) = "1"B
( "11"B = "110"B ) = ( "110"B = "110"B ) = "1"B
```

The result of a comparison expression whose operand includes '<' or '>' depends on the collating sequence. The collating sequence is a character string in which each possible character occurs exactly once. A character is less than another character if the first occurs before the second in the collating sequence. Similarly, a character is greater than another if the first occurs after the second. The collating sequnce is an unalterable part of the definition of GCOS PL/I, but it can vary from one implementation of PL/I to another.

The full collating sequence is given later in this section, under "The COLLATE Function". For the examples that are given here, an abbreviated version of the collating sequence is sufficient:

... (blank) ... 0123456789 ... ABCDEFGHIJKLMNOPQRSTUVWXYZ ...

Observe that, for example, C is less than M and, for another example, C is greater than the blank character.

Examples of "less than" and "greater than" comparison expressions are:

```
( "C" < "M" ) = "1"B
( "C" ^< "M" ) = "0"B
( "C" <= "M" ) = "1"B
```

When the operand values are of length greater than one and are not identical, they are examined from left to right until a difference is found; the first differing character position is the sole basis for the comparison. For example:

```
( "ABCDE" < "ABCXY" ) = ( "D" < "X" ) = "1"B
( "ABCDE" > "VWXYZ" ) = ( "A" > "V" ) = "0"B
```

As already noted, operands of different lengths are adjusted so that their lengths agree; for example:

```
( "ABC" < "ABCDE" ) = ("ABCbb" < "ABCDE" )
                    = ( "b" < "D" ) = "1"B

( "000" < "0000" ) = ( "000b" < "0000" )
                   = ( "b" < "0" ) = "1"B
```

When bit strings are compared, '0' is less than '1'; thus:

```
( "0"B < "1"B ) = "1"B
( "101101"B > "101111"B ) = ( "0"B > "1"B ) = "0"B
( "1"B < "100"B ) = ( "100"B < "100"B ) = "0"B
```

THE LENGTH FUNCTION

A reference to the function has the form:

```
LENGTH(S)
```

The result is a FIXED BIN(24,0) value that is the length of the string that is the value of S. For example,

```
LENGTH("ABCDE") = 5
LENGTH("") = 0
LENGTH("101010"B) = 6
```

Note that for a VARYING string the <u>length</u> is different from the <u>maximum</u> <u>length</u>. Suppose the variable X is declared by

```
DCL X CHAR(10) VARYING;
```

and has the value "ABC". Then

```
LENGTH(X) = 3
```

rather than 10.

THE REVERSE FUNCTION

A reference to the function has the form:

```
REVERSE(S)
```

The result is a string which is the reverse of the value of S. For example,

```
REVERSE("ABCDE") = "EDCBA"
REVERSE("A") = "A"
REVERSE("") = ""
REVERSE("11101"B) = "10111"B
```

Some of the built-in operations perform a left-to-right search of a string. When a right-to-left search is required, the REVERSE function can be used to adjust the given string. The following example finds the position of the first occurrence of A from the <u>right</u> end of the given string:

```
LENGTH("ABACAD") - INDEX(REVERSE("ABACAD"),"A") + 1
     = 6 - INDEX("DACABA","A") +1
     = 6 - 2 + 1
     = 5
```

Bit-String Operations

Certain built-in string operations can be applied only to bit strings; they are:

- The logical operators, which perform the "not", "and", and "inclusive or" operations of Boolean logic.
- The BOOL function, which is a generalization of the three logical operators and which can perform any of the 16 operations of Boolean logic.

The logical operators are used in all programming applications, especially when the operands are one-bit strings produced by tests of data. The BOOL function is used in occasional advanced applications, especially for masking of bit strings.

THE LOGICAL OPERATORS

A parenthesized expression for a logical operator has one of the forms:

```
(  ^  B2  )          (not)
(  B1  &  B2  )      (and)
(  B1  !  B2  )      (inclusive or)
```

The result is a bit-string value. When both operands are a bit string of length one, the result is given by one of the following examples:

```
( ^ "0"B ) = "1"B
( ^ "1"B ) = "0"B

( "0"B & "0"B ) = "0"B        ( "0"B ! "0"B ) = "0"B
( "0"B & "1"B ) = "0"B        ( "0"B ! "1"B ) = "1"B
( "1"B & "0"B ) = "0"B        ( "1"B ! "0"B ) = "1"B
( "1"B & "1"B ) = "1"B        ( "1"B ! "1"B ) = "1"B
```

When operand strings are longer than one bit, the rules just given are applied to the individual bits, as follows. For the "not" operation, the rule is applied to the first bit of the operand to produce the first bit of the result; and so on, for the other bit positions:

```
( ^ "1110001"B ) = "0001110"B
```

For the "and" and "or" operations, the rules are applied to the first bit of the first operand and the first bit of the second operand to produce the first bit of the result; and so on for the other bit positions:

```
( "1100"B & "1001"B ) = "1000"B
( "1100"B ! "1001"B ) = "1101"B
```

When one operand is shorter than the other, zero bits are added to the right end of the shorter operand; thus:

```
( "1"B & "0011"B ) = ( "1000"B & "0011"B ) = "0000"B
```

THE BOOL FUNCTION

A reference to the function has the form:

```
BOOL(B1,B2,B3)
```

The result is a bit string that is produced by applying to B1 and B2 and operation specified by B3. The third argument, B3, is converted to a bit string of length four. Suppose the bits in B3 are b1, b2, b3, and b4. Then if B1 and B2 are both bit strings of length 1, all possible results are given by the following examples:

```
BOOL( "0"B, "0"B, B3 )
BOOL( "0"B, "1"B, B3 )
BOOL( "1"B, "0"B, B3 )
BOOL( "1"B, "1"B, B3 )
```

As a specific example, suppose B3 is "0001"; then:

```
BOOL( "0"B, "0"B, "0001"B ) = "0"B
BOOL( "0"B, "1"B, "0001"B ) = "0"B
BOOL( "1"B, "0"B, "0001"B ) = "0"B
BOOL( "1"B, "1"B, "0001"B ) = "1"B
```

Comparison of these equations with the definition of the '&' operator given earlier suggests that they are the same; indeed:

```
BOOL(B1,B2,"0001"B) = ( B1 & B2 )
```

Similarly,

```
BOOL(B1,B2,"0111"B) = ( B1 ! B2 )
BOOL(B1,B2,"1010"B) = ( ^ B2 )
```

Thus the BOOL function provides a general way of defining logical operations.

When B1 and B2 are bit strings of length greater than one, they are operated upon on a bit-by-bit basis just as with the logical operators. For example:

```
BOOL("111"B,"101"B,"0001"B) = "101"B
```

When one operand is shorter than the other, zero bits are added to the right end of the shorter operand, just as with the logical operators; thus:

```
BOOL("1"B,"0011"B,"0001"B) = BOOL("1000"B,"0011"B,"0001"B) = "0000"B
```

Character-String Operations

Three built-in operations can be applied only to character strings; they are:

- The SEARCH and VERIFY functions, which are used to find the beginning and end of a generalized substring of a given string.
- The TRANSLATE function, which is used to modify a string by the systematic replacement of certain characters by certain other characters.

THE SEARCH FUNCTION

A reference to the function has the form:

```
SEARCH(C1,C2)
```

The result is a FIXED BIN(24,0) value that is the position in C1 of the leftmost occurrence of any character contained in C2. Sometimes C2 is a single character; for example:

```
SEARCH("+328.02",".") = 5
SEARCH("ALPHA,BETA,GAMMA",",") = 6
```

In other cases, more than one character is used in C2; thus:

```
SEARCH("A2*(BETA-GAMMA)","+-*/()") = 3
SEARCH("18.2344E-2","E+-") = 8
```

When C1 does not contain any character in C2, the result is zero:

```
SEARCH("+328.02",";") = 0
SEARCH("ABCDE","") = 0
```

THE VERIFY FUNCTION

A reference to the function has the form:

```
VERIFY(C1,C2)
```

The result is a FIXED BIN(24,0) value that is the position of the first character of C1 that does not occur in C2. For example:

```
VERIFY("ALPHA-BETA","ABCDEFGHIJKLMNOPQRSTUVWXYZ") = 6
VERIFY("1923.98E02","0123456789") = 5
```

When C1 contains only characters that are in C2, the result is zero:

```
VERIFY("1923.98E02","+-0123456789.E") = 0
```

The example just given suggests the reason for the name "verify"; the example checks a given string (which appears to be a decimal constant) to "verify" that it contains only allowed characters.

Although the function is used for "verification" as just described, it is more often used to "search for the end" of a given substring. As such, it is a complement to the SEARCH function, which is used to "search for the beginning" of a given substring. Suppose a procedure must be written to extract the leftmost substring that is a sequence of digits. Before writing the procedure, the programmer sketches it out as follows:

The given string might be:

```
"ALPHA*(238+BETA)"
```

The SEARCH function is used to locate the beginning of the desired substring:

```
SEARCH("ALPHA*(238+BETA)","0123456789") = 8
```

Application of the SUBSTR function gives

```
SUBSTR("ALPHA*(238+BETA)",8) = "238+BETA)"
```

The VERIFY function is used to locate the end of the desired substring:

```
VERIFY("238+BETA)","0123456789") = 4
```

Application of the SUBSTR function gives:

```
SUBSTR("238+BETA)",1,3) = "238"
```

which is the desired result.

THE TRANSLATE FUNCTION

A reference to the function has one of the forms:

```
TRANSLATE(C1,C2,C3)
TRANSLATE(C1,C2)
```

The result is a modification of C1 in which each character in C3 is replaced in C1 by the corresponding character in C2. For example:

```
TRANSLATE("ABCDE","2","B") = "A2CDE"
TRANSLATE("12345","ABCDEFG","1234567") = "ABCDE"
TRANSLATE("28.923E-18","000000000","0123456789") = "00.000E-00"
TRANSLATE("1","ABCD","1111") = "A"
```

C2 and C3 should be strings of equal length; however, if C2 is shorter than C3, blanks are added to the right end of C2:

```
TRANSLATE("ABCDE","B","BCDE")
    = TRANSLATE("ABCDE","Bøøøø","BCDE") = "ABøøøø"
```

Both C2 and C3 should be given (the second form is not recommended); however, if C3 is not given, the collating sequence is assumed:

```
TRANSLATE(C1,C2) = TRANSLATE(C1,C2,COLLATE())
```

Character-Set Operations

Three built-in string operations are relevant to the character set of GCOS PL/I. They are:

- The COLLATE function, which yields the string that gives the collating sequence of GCOS PL/I.
- The HIGH and LOW functions, which yield a sequence of control characters that have a special purpose in the preparation of output.

THE COLLATE FUNCTION

A reference to the function has the form

```
COLLATE                    COLLATE()
```

The result is a character string that gives the collating sequence for the characters that can be used in character-string values in PL/I. If a given character occurs before a second given character in the collating sequence, then the first character is "less than" the second. In this way, the collating sequence defines the relational operators as they apply to character strings.

The collating sequence is not the same in every implementation of PL/I; one reason for this variation is that different implementations use different character sets. GCOS PL/I uses the ASCII character set, and the collating sequence is the ASCII character set arranged according to the ASCII octal codes. The character with code 000 is first, the character with code 001 is next, and so on. It follows that the ASCII code for a character can be obtained by means of the INDEX function. For example, to obtain the decimal equivalent of the ASCII code for the character "A", write:

```
INDEX(COLLATE(),"A")-1 = 97     (octal code 141)
```

Similarly, to obtain the character whose decimal code is 97, write:

```
SUBSTR(COLLATE(),97+1,1) = "A"
```

There are 128 characters in the ASCII character set and therefore the length of the collating sequence is 128. The sequence can be described in four parts, as follows:

1. The first part is 32 nonprinting control characters, as follows:

```
000     NUL     (null character)
001
...         (unused)
006
007     BEL     (alarm)
010     BS      (backspace)
011     HT      (horizontal tab)
012     NL      (new line = carriage return and line feed)
013     VT      (vertical tab)
014     NP      (new page = carriage return and form feed)
015     CR      (carriage return)
016     RRS     (red ribbon shift)
017     BRS     (black ribbon shift)
020
...         (unused)
036
037     EGM     (enter graphic mode)
```

2. The second part is the blank character.

3. The third part is the 94 printing characters, as follows:

```
! " # $ % & ' ( ) * + , - . /

0 1 2 3 4 5 6 7 8 9

: ; < = > ? @

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

[ \ ] ^ _ `

a b c d e f g h i j k l m n o p q r s t u v w x y z

{ | } ~
```

4. The fourth part is a single nonprinting control character, as follows:

```
177     PAD     (padding character)
```

If words are alphabetized by means of the relational operators, then the collating sequence determines the order of the words. The important features of the collating sequence are:

- The blank character comes before any printing character; therefore the following ordering results:

```
PLAN
PLANE
```

- Any digit comes before any letter. Therefore the following ordering results:

```
ALPHA
ALPHA2
ALPHA3
ALPHABET
```

- The letters appear in the collating sequence in uninterrupted alphabetical order. Therefore the following ordering results:

  Boston
  Cambridge

- The punctuation marks are scattered throughout the collating sequence. Therefore they must be given special consideration in alphabetization.

THE LOW FUNCTION

A reference to the function has the form:

LOW(I)

The result is a character string composed of I "NUL" characters. The "NUL" character is the first character in the collating sequence.

THE HIGH FUNCTION

A reference to the function has the form:

HIGH(I)

The result is a character string composed of I "PAD" characters. The "PAD" character is the last character in the collating sequence.

String Functions Defined Elsewhere

The following built-in functions yield string values but are defined elsewhere in this section, as indicated, because they are specialized:

BIT CHARACTER STRING	UNSPEC VALID	See "Conversion Operations".
ONCHAR ONFIELD ONFILE	ONKEY ONLOC ONSOURCE	See "System Variable Operations".
DATE	TIME	See "System Variable Operations".

ADDRESS AND AREA OPERATIONS

The operations discussed here include operators for the comparison of address values, functions that have addresses as their values, and a single function that has an area as its value. All of these operations are considerably removed from the field of routine programming; they are certainly useful where they are needed and include some of the innovative features of PL/I; but they are rarely required outside of the storage management techniques that characterize advanced applications.

General Address Functions

There are five general functions and operations for the manipulation of address values; they are "general" in the sense that they are independent of the representation of address values in a particular implementation of PL/I. The functions are:

- The two relational operators '=' and '^=', which can be used for the comparison of address values
- The ADDR function, which yields a pointer value that is equivalent to any given variable reference
- The NULL and NULLO functions, which yield the special null locative value for a pointer or an offset.

The relational operators for address values are useful whenever file values, local values, and so on, must be manipulated; such applications occur in intermediate and advanced applications. The other functions discussed here are used primarily in advanced applications that use based variables, list processing, and storage sharing.

THE RELATIONAL OPERATORS FOR ADDRESS VALUES

A parenthesized operator expression with a relational operator has the form:

(A1 op A2)

where A1 and A2 must yield address values and the operator, op, is one of the following:

=	(is equal to)
^=	(is not equal to)

Relational operators can also apply to computational values, as described earlier under "The Relational Operators for Arithmetic Values" and "The Relational Operators for String Values".

The result of the relational expression is "1"B or "0"B, depending on whether the comparison is true or false. Recall that there are six types of address values, as follows:

LABEL:	the destination of a GOTO statement
ENTRY:	the destination of a CALL or a function reference
FORMAT:	the object of a remote format reference
POINTER:	the "absolute" address of a storage unit
OFFSET:	the "relative" address of a storage unit
FILE:	the address of a file-state block

With one exception, both arguments of an address relational operator must have the same type. The exception is the combination of a pointer value and an offset value; in this case, the offset value is converted to a pointer value.

THE ADDR FUNCTION

A reference to the function has the form:

```
ADDR(U)
```

where U must be a reference to a connected variable. A variable is connected unless it is an aggregate variable whose components are interleaved with the components of some other variable. The result is a pointer value to the storage unit that is designated by U. Suppose the following declaration applies:

```
DCL 01 A(2),
       02 B FLOAT,
       02 C DEC(6,2);
```

Examples of the function follow:

ADDR(A)	gives a pointer to storage for the entire array of structures
ADDR(A(2))	gives a pointer to the second element of the array of structures
ADR(A2).C)	gives a pointer to a scalar component of A
ADDR(A.C)	is invalid because A.C designates an unconnected variable

The variable A.C is unconnected because it is made up of two scalar variables, designated by A(1).C and A(2).C, that are not adjacent in storage. According to the section on "Storage Types", the components of A occur in storage in the sequence:

```
A(1).B     A(1).C     A(2).B     A(2).C
```

and thus A(2).B is between the two components of A.C.

If U is a major (level-one) CONTROLLED variable reference that is not currently allocated, then

ADDR(U) = null

and the reference is valid. If U is any other variable that is not currently allocated, then the reference is invalid.

THE NULL FUNCTION

A reference to the function has one of the forms:

NULL NULL()

The result is the null value of data type POINTER. Suppose P1 is a pointer variable. Then the statement

P1 = NULL;

assigns the null value to P1 and thus gives P1 a defined value that does not point to any storage unit. The variable can later be tested by a statement such as:

IF P1=NULL THEN GOTO L2;

Observe that a pointer variable can be in one of the following states:

- If no value has been assigned to the variable, then the value of the variable is undefined and any reference to that value is invalid.
- If the null value has been assigned to the variable, then the value can be assigned to another variable or compared to another locative value; but it cannot be used as a locator-qualifier because it does not designate a storage unit.
- If a non-null value has been assigned to the variable, then the value of the variable can be used for any purpose appropriate for a locative value.

THE NULLO FUNCTION*

A reference to the function has the form:

NULLO NULLO()

The result is the OFFSET value null. An offset variable is set to null to indicate that it does not point to any storage unit. The usage of NULLO is similar to that described under "The NULL Function".

The '*' means that this function is not part of Standard PL/I. The function is not implementation-dependent, but it must be replaced by other language in a program that is transported to some other implementation of Standard PL/I.

An Implementation-Dependent Address Function

There is a function that manipulates a pointer value in a way that depends upon the GCOS representation of pointers; this function is not, of course, part of Standard PL/I. It is:

- The ADDREL function, which generates a pointer to the Nth word after the word designated by a given pointer.

This function should be used only under special circumstances. It is appropriate mostly for systems programming.

THE ADDREL FUNCTION*

A reference to this function has the form:

```
ADDREL(P,N)
```

where P must yield a scalar POINTER value and N must yield either:

an integer value of no more than 35 bits, or a bit-string value of length 18.

The result is the pointer value defined by:

word address:	W+N, where W is the word address portion of the value of P
bit offset:	0

Thus the result pointer designates the Nth word after the word to which P points.

The '*' after the name of this function indicates that the function is not in Standard PL/I. A reference to this function makes a program dependent on the data representation of GCOS PL/I.

An Area Function

There is one function for area values. It is:

- The EMPTY function, which gives the value of an area in which no variable is currently allocated

This function is used in advanced applications where special storage management techniques are used.

THE EMPTY FUNCTION

A reference to the function has the form:

EMPTY EMPTY()

The result is the empty value of data type AREA. Suppose A is declared as an area variable; then the statement:

```
A = EMPTY;
```

can be used to assign the empty value to A and thus free all storage units currently allocated in A.

ARRAY OPERATIONS

Most of the functions and operators of PL/I can be applied to both scalar and array operands. However, each of the functions described here must have an array as its first operand, and it is in this sense that these are array functions.

Extent Functions

There are three functions that yield the values that describe a dimension of an array. They are:

- The LBOUND and HBOUND functions, which yield the bounds of a given dimension of an array
- The DIMENSION function, which yields the extent of a given dimension of an array

These functions are useful for operations on an array whose bounds are given by variable expressions. They are essential for determining bounds or extents of an array that is a procedure parameter with '*' extents; this case is discussed later, in the section on "Procedure Invocation".

THE LBOUND AND HBOUND FUNCTIONS

References to these functions have the forms:

```
LBOUND(A,N)
HBOUND(A,N)
```

where A must yield an array value and N must yield a scalar value that can be converted to a 17-bit integer. The result is a 24-bit integer whose value is the lower bound or upper bound, respectively, of the Nth dimension of A. As a basis for examples, consider the program:

```
P:   PROC;
        DCL A2(J,-1:K+2,0:3) FLOAT CONTROLLED;
        ...
        J = 6;
        K = 20;
        ALLOCATE A2;
        ... (Computation #1)
        END;
```

As Computation #1 begins, the functions have the following values:

```
LBOUND(A2,1) =  1          HBOUND(A2,1) =  6
LBOUND(A2,2) = -1          HBOUND(A2,2) =  22
LBOUND(A2,3) =  0          HBOUND(A2,3) =  3
```

If N does not designate a declared dimension of A, a reference to either of these functions is invalid.

THE DIMENSION FUNCTION

A reference to the function has the form:

```
DIMENSION(A,N)          DIM(A,N)
```

where A must yield an array value and N must yield a scalar value that can be converted to a 17-bit integer. The result is the extent of the Nth dimension of A. For any valid arguments, A and N,

```
DIM(A,N) = HBOUND(A,N) - LBOUND(A,N) + 1
```

Special Array Functions

There are three functions for special operations on arrays. Each corresponds to a well-known mathematical operation, as follows:

- The SUM function, which performs the summation operation and is represented in mathematics by a capital sigma.
- The PROD function, which performs the product operation and is represented in mathematics by a capital pi.
- The DOT function, which performs the dot product operation and is represented in mathematics by a dot written between two vector names.

These functions are usually applied to FLOAT values, and their use in that context is simple. However, they are also defined for FIXED values.

THE SUM AND PROD FUNCTION

References to the functions have the forms:

```
SUM(A)
PROD(A)
```

where A must yield an array whose elements are arithmetic values. The result is a scalar value that is the sum or product of the elements of A. The data type of the result is the same as that of the argument, except for the precision of a fixed-point value. If A is FIXED with precision (p,q), then the precision of the result is

```
(71,q)     for BINARY base
(59,q)     for DECIMAL base
```

As a basis for discussion, consider the program:

```
P:   PROC;
       DCL A3(2,3) FLOAT DEC(5);
       DCL A4(3) FIXED DEC(6,2);
       ...
       A3 = 5;
       A3(1,3) = 10;
       A4(1) = 6;
       A4(2) = .5;
       A4(3) = .04;
       ... (Computation #1)
       END;
```

As Computation #1 begins:

```
SUM(A3) = 00035E0
SUM(A4) = 0...06.54      (59 digits)

PROD(A3) = 31250E0
PROD(A4) = 0...0.12      (59 digits)
```

THE DOT FUNCTION

A reference to the function has one of the forms:

```
DOT(A1,A2,P)
DOT(A1,A2,P,Q)
```

where A1 and A2 must each yield a one-dimensional array whose elements are arithmetic values, and where P and Q cannot be general expressions but instead must be decimal integer constants (Q can be signed). The precision of the converted values of A1 and A2 is given by:

Reference	Scaling	Precision
DOT(A1,A2,P)	FIXED	(P,0)
	FLOAT	(P)
DOT(A1,A2,P,Q)	FIXED	(P,Q)

The result is the dot product of A1 and A2; that is,

```
A1(M1)*A2(M2) + A1(M1+1)*A2(M2+1) + ... + A1(M1+E-1)*A2(M2+E-1)
```

where

```
M1 = LBOUND(A1,1)
M2 = LBOUND(A2,1)
E  = DIM(A1,1)
```

The operands must satisfy the condition:

```
DIM(A1,1) = DIM(A2,1)
```

CONVERSION OPERATIONS

With the exception of the special conversion functions, every conversion function given here can be interpreted by determining the data type of the target and then referring to the section on "Value Conversion" for a description of the required conversion. Thus although many examples of conversion are shown here, the rules for conversion of values are not given here.

The conversion functions of PL/I are both redundant and incomplete. There are three different ways to perform most, but not all, conversions; yet, for some important conversions, there is no way to perform the conversion that is acceptable to both GCOS PL/I and Standard PL/I. This situation is discussed later, under "Guidelines for Conversion Functions", and suggestions are given for the choice of a conversion function for various situations.

The designers of GCOS PL/I took the view that a conversion between the three major types of computational value,

arithmetic,
character-string, and
bit-string

should be performed explicitly by means of a conversion function. Therefore, the compiler provides a warning message whenever such a conversion is performed implicitly. In compensation, to make explicit conversion easier, the designers added the nonstandard CONVERT function, whose interpretation is much simpler than that of the standard conversion functions.

The Fundamental Conversion Function

There is one function that can perform all of the conversions between scalar values that are possible in PL/I. It is:

- The CONVERT function, which converts a given value to the data type of a given variable.

A simple and safe policy for conversion is to use only the CONVERT function for all explicit conversion. A more advanced policy is described under "Guidelines for Conversion".

THE CONVERT FUNCTION*

A reference to the function has the form:

CONVERT(U,V)

where U must be a reference to a scalar variable and V must yield a scalar value. The result of the function reference is the value of V converted to the data type of U. The function can be used to establish any data type as the target for conversion, provided the conversion is valid in PL/I. The set of valid conversions is described earlier, in the section on "Value Conversion".

A specialized but important application of this function arises when a value of one major type must be assigned to a variable of a different major type. Suppose, for example, that the following declarations apply:

```
DCL ALPHA FLOAT;
DCL BETA CHAR(20);
```

Within the scope of these declarations, the assignment statement

```
ALPHA = BETA;
```

is marked by a warning message by the GCOS PL/I compiler because it requires an implicit conversion between major types. On the other hand, the assignment statement

```
ALPHA = CONVERT(ALPHA,BETA);
```

is not marked because the conversion is explicit. Conversion between major types is usually considered to be important enough to merit an assignment statement of its own (rather than being performed with an expression); therefore, the usage just shown is common.

In order to consider the function in a general way, suppose the following declarations apply:

```
DCL FIXDEC62 FIXED DEC(6,2) BASED;

DCL S CHAR(12);
```

Within the scope of these declarations:

```
CONVERT(FIXDEC62,00028.3356) = 0028.33
CONVERT(FIXDEC62,55528.3356) = (SIZE)

CONVERT(FIXDEC62,"-28") = -0028.00

CONVERT(FIXDEC62,"-1111.001B") = -0015.12

CONVERT(S,"123456789012") = "123456789012"
CONVERT(S,"1234567890123") = (STRINGSIZE)

CONVERT(S,100) = "ßßß100ßßßßßß"

CONVERT(S,-5.1749E-2) = "-5.1749E-002"
```

The only purpose of the variable reference that is the first argument of CONVERT is to supply a data type; the <u>value</u> of that variable is neither used nor set. The variable can be declared especially for use in CONVERT or it can be a variable that is used for other purposes as well. If the variable is especially declared, it should be BASED so that no storage will be allocated for it.

The '*' means that this function is not part of Standard PL/I. A use of this function does not perform an operation that cannot be performed in Standard PL/I, but a program that uses CONVERT must be slightly changed to make it Standard. More is said of this under "Guidelines for Conversion Functions".

Conversion to Arithmetic Data Types

There are seven functions for conversion to an arithmetic data type, one for each of the attributes that are used in an arithmetic data type plus the IMAG function. They are:

- The REAL and COMPLEX functions for mode conversion
- The FIXED and FLOAT functions for scale conversion
- The BINARY and DECIMAL functions for base conversion
- The PRECISION function for precision conversion

These functions can be used individually, but they should not be compounded, as in

```
X = FLOAT(BINARY(Y))
```

because the result is difficult to predict and often is not what is desired.

THE REAL FUNCTION FOR CONVERSION

A reference to this function has the form:

```
REAL(Z)
```

where Z must be COMPLEX. The result is a REAL value that is the value of the real part of Z. Except for the change in the mode attribute, the storage type of the result is the same as the storage type of Z. If the imaginary part of the value of Z is zero, then the result has the same mathematical value as Z, and the function therefore performs a conversion from COMPLEX to REAL mode. For example:

```
REAL(-6.0000E3+0.0000E0I) = -6.0000E3

REAL(.11100011E8B+.000000000E0BI) = .11100011E8B
```

If the imaginary part is not zero, the function does not perform a pure conversion operation, but rather an operation of complex arithmetic as discussed earlier under "The REAL and IMAG Functions for Arithmetic".

THE COMPLEX FUNCTION FOR CONVERSION

A reference to this function has the form:

```
COMPLEX(R,0)
```

where R must be REAL. The result is a COMPLEX value whose real part is the value of R and whose imaginary part is zero. The result has the same mathematical value as R, and the function (in this form) therefore performs a conversion from REAL to COMPLEX. Complex computations are usually performed with FLOAT values; and for such values the storage type of the result is the same as the storage type of R except for the desired change of mode. Examples are:

```
COMPLEX(-6.0000E3,0) = -6.0000E3+0.0000E0I

COMPLEX(.111000111E8B,0) = .111000111E8B+.000000000E0BI
```

When the second argument of the COMPLEX function is not zero, the function does not perform a pure conversion operation, but rather an operation of complex arithmetic as discussed earlier under "The COMPLEX Function for Arithmetic".

THE FIXED FUNCTION

A reference to this function has one of the forms:

```
FIXED(X)
FIXED(X,P)
FIXED(X,P,Q)
```

where X must yield an arithmetic value and where P and Q cannot be general expressions but instead must be given as decimal integer constants (Q can be signed). The result is the value of X converted to a certain storage type. The storage type of the result is determined as follows:

Reference	Data Type of X	Data Type of Result
FIXED(X)	mode FIXED base(p,q)	mode FIXED base(p,q)
	mode FLOAT base(p)	mode FIXED base(p,0)
	CHAR(m)	REAL FIXED DEC(59,0)
	BIT(m)	REAL FIXED BIN(71,0)
FIXED(X,P)	...	as above, but with precision (P,0)
FIXED(X,P,Q)	...	as above, but with precision (P,Q)

Examples are:

```
FIXED(-0435.241)      = -0435.241
FIXED(-0435.241,5,2) = -435.24
FIXED(-0435.421,4,2) = (SIZE)
FIXED(-0435.421,4)    = -0435.

FIXED(+000111000.000B,9,3) = +111000.000B

FIXED(+000243.E2) = +024300.

FIXED("-5.823E2")      = -0...582.       (59 digits)
FIXED("-5.823E2",7,2) = -00582.30

FIXED("110B",4,1) = 006.0
FIXED("110"B,4,1) = 110.0B

FIXED(-81143.E-2+68134.E-71,3) = -811.+000.1
```

THE FLOAT FUNCTION

A reference to this function has one of the forms:

```
FLOAT(X)
FLOAT(X,P)
```

where X must yield an arithmetic value and where P cannot be a general expression but instead must be given as a decimal integer constant. The result is the value of X converted to a certain storage type. The scaling attribute of the result is determined as follows:

Reference	Data Type of X	Data Type of Result
FLOAT(X)	mode FIXED base(p,q)	mode FLOAT base(p)
	mode FLOAT base(p)	mode FLOAT base(p)
	CHAR(m)	REAL FLOAT DEC(59)
	BIT(m)	REAL FLOAT BIN(63)
FLOAT(X,P)	...	as above, but with precision (P)

Examples are:

```
FLOAT(-89241E3)    = -89241E3
FLOAT(-89241E3,3) = -892E5

FLOAT(+111000111000E4,12) = +111000111000E4

FLOAT("ØØ200ØØØØ")    = 0...200E0      (59 mantissa digits)
FLOAT("ØØ200ØØØØ",5) = 00200E0
```

THE BINARY FUNCTION

A reference to this function has one of the forms:

BINARY(X)	BIN(X)
BINARY(X,P)	BIN(X,P)
BINARY(X,P,Q)	BIN(X,P,Q)

where X must yield an arithmetic value and where P and Q cannot be general expressions but instead must be given as decimal integer constants (Q can be signed). The result is the value of X converted to a certain storage type that is determined as follows:

Reference	Data Type of X	Data Type of Result
BIN(X)	mode FIXED BIN(p,q)	mode FIXED BIN(p,q)
	mode FIXED DEC(p,q)	mode FIXED BIN(p1,q1) p1 = CEIL(p*3.32)+1 q1 = CEIL(q*3.32) for q >= 0 q1 = -CEIL(-q*3.32) for q < 0
	mode FLOAT BIN(p)	mode FLOAT BIN(p)
	mode FLOAT DEC(p)	mode FLOAT BIN(p1) p1 = CEIL(p*3.32)
	CHAR(m)	REAL FIXED BIN(71,0)
	BIT(m)	REAL FIXED BIN(71,0)
BIN(X,P)	...	as above, but with precisions (P,0) for a FIXED result and (P) for a FLOAT result
BIN(X,P,Q)	(this reference is not valid if X is FLOAT)	as above, but with precision (P,Q)

Examples are:

```
BIN(-1110111.0111B)    = -1110111.0111B
BIN(-1110111.0111B,8) = -01110111B

BIN(28.25) = 00011100.0100000B

BIN(-.11101110111E8B) = -.11101110111E8B

BIN("28.25",12,6) = 011100.010000B

BIN("111000111"B,12,2) = 0111000111.00B
```

THE DECIMAL FUNCTION

A reference to this function has one of the forms:

```
DECIMAL(X)          DEC(X)
DECIMAL(X,P)        DEC(X,P)
DECIMAL(X,P,Q)      DEC(X,P,Q)
```

where X must yield an arithmetic value and where P and Q cannot be general expressions but instead must be given as decimal integer constants (Q can be signed). The result is the value of X converted to a certain storage type that is determined as follows:

Reference	Data Type of X	Data Type of Result
DEC(X)	mode FIXED BIN(p,q)	mode FIXED DEC(p1,q1) p1 = CEIL(p/3.32) + 1 q1 = CEIL(q/3.32) for q >= 0 q1 = -CEIL(-q/3.32) for q < 0
	mode FIXED DEC(p,q)	mode FIXED DEC(p,q)
	mode FLOAT BIN(p)	mode FLOAT DEC(p1) p1 = CEIL(p/3.32)
	mode FLOAT DEC(p)	mode FLOAT DEC(p)
	CHAR(m)	REAL FIXED DEC(59,0)
	BIT(m)	REAL FIXED DEC(59,0)
DEC(X,P)	...	as above, but with precisions (P,0) for a FIXED result and (P) for a FLOAT result
DEC(X,P,Q)	(this reference is not valid if X is FLOAT)	as above, but with precision (P,Q)

Examples are:

```
DEC(-0435.241)      = -0435.241
DEC(-0435.241,5,3) = (SIZE)

DEC(1111.110000B) = 015.75

DEC(7.8100E0,7) = 7.810000E0

DEC("28.25",6,3) = 028.250

DEC("110.110B",4,3) = 6.750
```

THE PRECISION FUNCTION

A reference to this function has one of the forms:

PRECISION(X,P)	PREC(X,P)
PRECISION(X,P,Q)	PREC(X,P,Q)

where X must yield an arithmetic. value and where P and Q cannot be general expressions but instead must be given as decimal integer constants (Q can be signed). The result is the value of X converted to a certain storage type that is determined as follows:

Reference	Data Type of X	Data Type of Result
PREC(X,P)	mode FIXED base(p,q)	mode FIXED base(P,0)
	mode FLOAT base(p)	mode FLOAT base(P)
	CHAR(m)	REAL FIXED DEC(P,0)
	BIT(m)	REAL FIXED BIN(P,0)
PREC(X,P,Q)	(this reference is not valid if X is FLOAT)	as above, but with precision (P,Q)

Examples are:

```
PREC(18.7809,5,1) = 0018.7

PREC(18.7809,5) = 00018

PREC(-111000111.11B,10) = -0111000111B

PREC(-8.2315E0,7) = -8.231500E0
```

Conversion to String Data Types

There are two functions for conversion to string data types. They are:

- The CHAR function, which converts a value to a character string.
- The BIT function, which converts a value to a bit string.

THE CHARACTER FUNCTION

A reference to this function has one of the forms:

CHARACTER(S)	CHAR(S)
CHARACTER(S,I)	CHAR(S,I)

where S must yield an arithmetic value and I must be a scalar value.that can be converted to a 24-bit integer. The result is a character string whose maximum length is determined as follows:

Reference	Data Type of Result
CHAR(S)	CHAR(*) NONVARYING
CHAR(S,I)	CHAR(I) NONVARYING

The '*' means "exactly enough characters to represent the value of S after it has been converted to a CHARACTER value". Examples are:

```
CHAR("ABCDE")    = "ABCDE"
CHAR("ABCDE",7) = "ABCDEØØ"
CHAR("ABCDE",3) = (STRINGSIZE)

CHAR("11011"B)    = "11011"
CHAR("11011"B,7) = "11011ØØ"
CHAR("11011"B,3) = (STRINGSIZE)

CHAR(+81993E6)     = "Ø8.1993E+010"
CHAR(+81993E6,14) = "Ø8.1993E+010ØØ"
CHAR(+81993E6,10) = (STRINGSIZE)

CHAR(-0820)     = "ØØØ-820"
CHAR(-0820,12) = "ØØØ-820ØØØØØ"
CHAR(-0820,6)   = (STRINGSIZE)

CHAR(.1011001...E7B,20) = "Ø8.90000000E+001ØØØØ"
```

In the last example, "..." represents enough zeros to make the argument FLOAT(27).

THE BIT FUNCTION

A reference to this function has one of the forms:

```
BIT(S)
BIT(S,I)
```

where S must yield an arithmetic value and I must yield a scalar value that can be converted to a 24-bit integer. The result is a bit string whose maximum length is determined as follows:

Reference	Data Type of Result
BIT(S)	BIT(*) NONVARYING
BIT(S,I)	BIT(I) NONVARYING

The '*' means "exactly enough characters to represent the value of S after it has been converted to a BIT value". Examples are:

```
BIT("11011"B)    = "11011"B
BIT("11011"B,7) = "1101100"B
BIT("11011"B,3) = (STRINGSIZE)

BIT("11011") = "11011"B
BIT("11012") = (CONVERSION)

BIT(.1101111E5B) = "0011100"B
BIT(18.983E0)    = "0000000000010011"B

BIT(-065.29)      = "0001000001"B
BIT(-065.29,12)  = "000100000100"B
```

Conversion between Locative Data Types

There are two functions for conversion between the two locative data types. They are:

- The standard POINTER function, which converts an offset value to a pointer.
- The OFFSET function, which converts a pointer value to an offset.

These functions are used when locative values are used in connection with AREA values.

THE POINTER FUNCTION

A reference to the function has the form:

POINTER(X,A) PTR(X,A)

where X must yield a scalar offset value and A must yield a scalar area value. X must designate a storage unit that is currently allocated in the area A; and the result is a POINTER value that designates the same storage unit. Thus the function converts an OFFSET value to a POINTER value.

THE OFFSET FUNCTION

A reference to the function has the form:

OFFSET(P,A)

where P must yield a scalar pointer value and A must yield a scalar area value. P must designate a storage unit that is currently allocated in the area A; and the result is an OFFSET value that designates the same storage unit. Thus the function converts a POINTER value to an OFFSET value.

Special Conversion Functions

There are three special functions for conversion. They are:

- The STRING function, which converts a packed aggregate of string values into a scalar string value.
- The UNSPEC function, which converts any value into a bit string that represents its actual representation in memory.
- The VALID function, which checks, after the fact, whether or not a given string can be assigned to a given pictured variable.

These functions are used in advanced applications that use storage sharing and special input/output.

THE STRING FUNCTION

A reference to the function has the form:

```
STRING(U)
```

The function reference is interpreted as follows:

- If U yields a scalar bit-string value, the result is U.
- If U yields a scalar value of any other computational type, the value is converted to CHAR(*) and the converted value is the result. The '*' means "exactly enough characters to represent the value of U when it has been converted to a CHARACTER value".
- If U yields an aggregate value whose components are unaligned, nonvarying, bit-string variables, the result is a scalar bit-string value that is the concatenation of all of the components of the value of U.
- If U yields an aggregate value whose components are unaligned, nonvarying, character-string or numeric-pictured variables, the result is a scalar character-string value that is the concatenation of all of the components of the value of U.

If none of these cases apply, then the reference is invalid. This is the only conversion function that can convert an aggregate value to a scalar value in an implementation-independent way. It is very useful in certain specialized applications. For example, consider the following program:

```
DCL 01 MEMBER UNALIGNED,
          02 NAME CHAR(30),
          02 ADDRESS CHAR(30),
          02 CITYSTATE CHAR(30);
     DCL SMEM CHAR(90);
     ...
     SMEM = STRING(MEMBER);
     ...
     END;
```

In this program NAME, ADDRESS, and CITYSTATE can be manipulated as individual character strings, but when it is useful to have them as a single string, they can be assigned to SMEM through the STRING function. The assignment statement is equivalent to:

```
SMEM = NAME!!ADDRESS!!CITYSTATE;
```

The STRING function is closely related to string overlay defining which is discussed earlier, in the section on "Storage Management".

The pseudo-variable named STRING is described later, in the section on "Value Assignment".

THE UNSPEC FUNCTION

A reference to the function has the form:

```
UNSPEC(U)
```

where U must be a reference to a variable. The result is a bit-string value that is the GCOS internal representation of the value of U, as described in the section on "Storage Types". Consider, for example, the program:

```
P:   PROC;
        DCL 01 ITEM UNALIGNED,
               02 COUNT FIXED,
               02 CODE CHAR(2),
               02 RATE DEC(2,1);
        ...
        COUNT = 67;
        CODE = "CC";
        RATE = -0.5;
        ...
        END;
```

After the three assignment statements, the values of the UNSPEC function are:

```
UNSPEC(COUNT) = "0000000000001000011"B
UNSPEC(CODE)  = "XX1000011XX1000011"B
UNSPEC(RATE)  = "XX0101101XX0110000XX0110101"B

UNSPEC(ITEM)  = "0000000000001000011XX1000011XX1000011
                 XX0101101XX0110000XX0110101"B
```

The XX is used for the two high-order bits of each character code because these bits are reserved in GCOS. This example uses a variable that is UNALIGNED, but (in contrast to the STRING function) there is no restriction on the variable mentioned in the UNSPEC function. If this example were repeated without the UNALIGNED attribute, UNSPEC(ITEM) would contain 108 bits instead of 63 bits.

One use of the UNSPEC function is as follows: the contents of a variable is converted to a bit-string by means of the UNSPEC function and the result is output as a record to some storage device; later, the string is input and is assigned back to a variable of exactly the same storage type by means of the UNSPEC pseudo-variable. This use is a valid and standard use of PL/I, because it produces the same result regardless of the particular internal representation of a given implementation. Other uses, which test or manipulate the value of UNSPEC are not valid in Standard PL/I because they are, of course, implementation-dependent.

The pseudo-variable named UNSPEC is described later, in the section on "Value Assignment".

THE VALID FUNCTION

A reference to the function has the form:

```
VALID(S)
```

where S must be a scalar pictured variable. The result is "1"B or "0"B, depending on whether or not the value contained in S can be edited into the picture declared for S. The function is used in connection with storage sharing to check to see if an invalid value has been assigned to the pictured variable by way of an equivalenced variable that does not have a picture. For example, consider the program:

```
P:   PROC;
        DCL S1 PIC"9999";
        DCL S2 CHAR(4) BASED;
        DCL P POINTER;
        P = ADDR(S1);
        ...
        P->S2 = "-500";
        ... (Computation #1)
        END;
```

This program sneaks an invalid character-string value into the pictured numeric variable S1; that is, an assignment is made to P->S2 when P points to S1. If the assignment had been written as:

```
S1 = "-500";
```

then the CONVERSION condition would have occurred. The VALID function can be used to check for such invalid assignments. The value of the function as Computation #1 begins is:

```
VALID(S1) = "0"B
```

This indicates that the value of S1, a sign and three decimal digits is not consistent with the picture, four decimal digits.

Guidelines for Conversion Functions

For many data type conversions, there are three ways to effect the conversion:

- assignment to a variable of the desired storage type and related forms of implicit conversion
- use of one of the standard conversion functions; that is, the functions whose names coincide with data type attributes, such as FIXED
- use of the nonstandard CONVERT function

Suggestions for a choice from among these possibilities are given here.

If the target of the assignment is arithmetic, then special care must be exercised because the compounding of some standard conversion functions produces results which are undesirable. Therefore:

1. When a conversion can be effected by a single standard conversion function, that function should be used.

2. When a conversion can be effected by a mode function compounded with a single scaling, base, or precision function, then those functions should be used.

3. When neither Rule 1 nor Rule 2 applies:

   a. If it is unlikely that the program will be transported from GCOS to some other PL/I implementation, the CONVERT function should be used.

   b. If transportation of the program is likely, the implicit conversion should be used. If GCOS generates a warning message because of the implicit conversion, the message should be ignored.

If the target of the assignment is character-string or bit-string, then the standard functions should be used in preference to the CONVERT function because they produce satisfactory results without departing from the standard.

Some GCOS programmers, who are not concerned about a departure from the standard, make consistent use of the CONVERT function to indicate all major conversions in a uniform way. In this case, the choice is a matter of programming style.

SYSTEM VARIABLE OPERATIONS

The PL/I interpreter maintains certain variables that cannot be accessed directly by a program. The purpose of most of the built-in functions described here is to provide a restricted form of access to these variables.

For example, there is a system variable that contains an encodement of the time of day, and this value can be retrieved by the TIME built-in function but cannot be set by an ordinary PL/I program. For another example, there is a system variable for each PRINT file that is open that contains the current page number of the file, and this value can be retrieved by the PAGENO function and set by the PAGENO pseudo-variable.

Several functions included here do not access system variables, but those functions are highly specialized, nonstandard functions that do depend heavily on the implementation of PL/I.

System Counter Functions

There are four functions that take their values from counters maintained by the PL/I interpreter. They are:

- The LINENO and PAGENO functions, which yield the appropriate integer values for a given PRINT file.
- The TIME and DATE functions, which yield appropriate string values.

These functions are of general interest.

THE LINENO AND PAGENO FUNCTIONS

References to these functions have the forms:

```
LINENO(F)
PAGENO(F)
```

where F must yield a scalar file value that has the PRINT attribute. The result is a 35-bit integer that is the current line number or page number, respectively, of the file F. The value is obtained from the file-state block designated by F.

The pseudo-variable named PAGENO is described later, in the section on "Value Assignment".

THE TIME AND DATE FUNCTIONS

References to these functions have the forms:

```
TIME                    TIME()
DATE                    DATE()
```

The result of the TIME function is a character string of length 12 whose value is

```
2 characters for the hour
2 characters for the minute
2 characters for the second
6 characters for the microseconds
```

The result of the DATE function is a character string of length six whose value is

2 characters for the year
2 characters for the month
2 characters for the day

For example,

TIME = 220318730211
DATE = 740703

is interpreted as

10:03:18.730211 pm
July 3, 1974

The Storage Management Functions

There are two functions that are specifically designed for storage management. They are:

- The ALLOCATION function, which yields the current number of allocations for a CONTROLLED variable.
- The SIZE function, which yields the number of GCOS words necessary for a given variable.

These functions are used in advanced applications where special storage management techniques are required.

THE ALLOCATION FUNCTION

A reference to the function has the form:

ALLOCATION(U) ALLOCN(U)

where U must be a major (level-one) controlled variable reference. The result is a 17-bit integer that is the number of generations of U that are currently allocated. Consider the program.

```
P:   PROC;
        DCL A(100) CHAR(20) CONTROLLED;
        ... (Computation #1)
        ALLOCATE A;
        ...(Computation #2)
        ALLOCATE A;
        ...(Computation #3)
        FREE A;
        FREE A;
        ...(Computation #4)
        END;
```

The value of ALLOCATION(A) is

0, 1, 2, and 0

during Computations #1, #2, #3, and #4, respectively.

THE SIZE FUNCTION*

A reference to this function has the form:

SIZE(U)

where U must be a simple reference to a major (level-one) variable. The result is a 24-bit integer whose value is the number of words that would be necessary to allocate a storage unit for U at the time the SIZE function is evaluated. When U is declared with variable extent expressions, the value of this function depends on those expressions. For example, consider the following declarations:

```
DCL S CHAR(2*I) CONTROLLED;
DCL 01  X BASED,
        02 N FIXED,
        02 A(I+1 REFER(N)) FLOAT;
DCL I FIXED;
```

If I = 10, then

```
SIZE (S) = 5
SIZE (X) = 12
SIZE (I) = 1
```

The '*' after the name of this function indicates that the function is not Standard PL/I. Furthermore, a reference to this function makes a program dependent on the data representation of GCOS PL/I.

The ON Condition Functions

When the PL/I processor detects an exceptional condition, it invokes an ON unit. This action is, in effect, an invocation of a procedure without parameters; instead, the necessary data is communicated through system variables. These system variables are accessed by the ON condition built-in functions.

For example, the exceptional condition KEY occurs when an attempt is made to input a keyed record that does not exist. To support the processing of this condition, the key (a character string) for which no record could be found is placed in a system variable provided for the purpose. The ON unit that processes the condition can use the built-in function ONKEY to retrieve the value of this system variable.

Every system variable associated with a condition is, in fact, a stack of variables and thus resembles a CONTROLLED variable. Each time a condition occurs and is signalled a new variable is allocated on each associated stack and its value is set. Each time the handling of a condition is complete, the most recent variable on each associated stack is freed and its value is lost. PL/I provides stacks for condition parameters because condition handling can be recursive; that is, a condition can occur while a previous occurrence of the same condition is being handled.

Before execution of a program begins, the PL/I processor allocates a variable for each of the system variables under discussion. This variable is set to a single blank character for the ONCHAR, a zero for ONCODE, and a null string for other ON condition functions. When an ON condition is not currently being handled, the associated system variables yield the values just mentioned.

The ON condition built-in functions are meaningful only in the context of the ON conditions themselves; these are discussed later, under "Condition Handling". The whole subject is relevant only to intermediate or advanced programming applications, where the user cannot simply make use of the default handling of conditions provided by PL/I.

THE ONLOC FUNCTION

A reference to the function has the form:

ONLOC ONLOC()

The value of this function is set whenever any condition occurs. When the condition is signalled, a character-string value that designates the most recently entered procedure block is placed on the stack associated with the ONLOC function. Suppose, for example, the following program is executed:

```
P:   PROC;
        DCL Y FLOAT;
INV:       PROC(X) RETURNS(FLOAT);
           DCL X FLOAT;
           RETURN(1/X);
           END;
        ...
        Y = INV(0);
        ...
        END;
```

When the division is performed, the ZERODIVIDE condition occurs and is signalled; and the value of the function is:

ONLOC = "INV"

When no condition is being handled, the value of the function is:

ONLOC = "" (the null string)

THE ONCODE FUNCTION

A reference to the function has the form:

ONCODE ONCODE()

The value of this function is set whenever any PL/I condition occurs. When the condition is signalled, a 35-bit integer that indicates the cause of the condition is placed on the stack associated with the ONCODE function. When no condition is being handled, the value of the function is

ONCODE = 0

Because the run-time subroutines that support the execution of PL/I programs are subject to modification and improvement, the list of error code values is subject to change and is not published in this document. Even when these codes are published, a program whose logic depends on a value of ONCODE may not run properly on other implementations of PL/I or on future versions of GCOS PL/I. Generally, the only valid use of the value of ONCODE is as part of an error message.

THE ONKEY FUNCTION

A reference to the function has the form:

ONKEY ONKEY()

The value of this function is set whenever a keyed input/output operation is in progress and one of the following conditions is signalled for that operation:

ENDFILE KEY RECORD TRANSMIT

When one of these conditions is signalled, the key given in the input/output statement is placed on the stack associated with ONKEY. Consider, for example, the statement:

```
READ FILE(ALPHA) KEY("BETA");
```

Suppose that file ALPHA contains no record whose key is BETA. Then the key condition occurs and

```
ONKEY = "BETA"
```

When no KEY condition is being handled, the value of the function is:

```
ONKEY = ""      (the null string)
```

THE ONFIELD FUNCTION

A reference to the function has the form:

ONFIELD ONFIELD()

The value of this function is set when the NAME condition is signalled. The NAME condition occurs during data-directed stream input/output when a name is encountered that is not mentioned in the list in the input statement. When the condition is signalled, the character string just extracted from the input stream is placed on the ONFIELD stack. Suppose, for example, the statement

```
GET FILE(SYSIN) DATA(ALPHA,BETA,GAMMA);
```

is executed when the input stream is

```
ALPHA=28.3,BETTA=61.4,GAMMA=19.2;
```

then the NAME condition is signalled and the value of the function is

```
ONFIELD = "BETTA=61.4"
```

When no NAME condition is being handled, the value of the function is:

```
ONFIELD = ""      (the null string)
```

THE ONCHAR AND ONSOURCE FUNCTIONS

References to these functions have the forms:

ONCHAR	ONCHAR()
ONSOURCE	ONSOURCE()

The value of ONSOURCE is the character string that is on the top of the stack associated with the ONSOURCE function. The value of ONCHAR is a single character, the conversion character in the character string that is on the top of the stack associated with the ONSOURCE function.

The ONSOURCE and ONCHAR functions are associated with the CONVERSION condition. The CONVERSION condition occurs when an attempt is made to convert a character-string value to an arithmetic or pictured value and the attempt fails because the given character-string value has the wrong form. The conversion character is defined as follows:

- If the given character-string can be corrected only by changing some characters, then the conversion character is the leftmost character that must be changed in any correction of the given character-string.
- If the given character-string can be corrected simply by adding some characters at the end, then the conversion character is the last character of the given character-string.

When no CONVERSION condition is being handled, the values of the functions are:

```
ONSOURCE = ""
ONCHAR   = " "
```

As a basis for an example of these functions, consider the following assignment statement:

```
X = FLOAT(Y);
```

Suppose that Y is a character-string variable and its value is:

```
"-12300Z5"
```

When the statement is executed and an attempt is made to evaluate the FLOAT built-in function, the processor finds that the string "-12300Z5" is not a valid representation of an arithmetic value. When the CONVERSION condition occurs,

```
ONSOURCE = "-12300Z5"
ONCHAR = "Z"
```

Now suppose that the value of Y is:

```
"-12300E+"
```

When the assignment statement is executed, the processor finds that the string is a valid beginning for a representation of an arithmetic value; for example, it could be corrected by adding the character '5'. Therefore,

```
ONSOURCE = "-12300E+"
ONCHAR = "+"
```

Finally, suppose that the value of Y is:

```
"-12300+5"
```

Once again, the processor finds that the given character string is not a valid representation for an arithmetic value. A human reader might say that the E that separates the mantissa from the exponent is missing, so that the value of ONCHAR should be '+'. However, the PL/I processor recognizes that the representation can be corrected by adding an I at the end to produce a representation of a COMPLEX value. Therefore,

```
ONSOURCE = "-12300+5"
ONCHAR = "5"
```

The pseudo-variables ONSOURCE and ONCHAR are described later, in the section on "Value Assignment".

THE ONFILE FUNCTION

A reference to this function has the form:

```
ONFILE          ONFILE()
```

The value of this function is set whenever an input/output operation is in progress and one of the following conditions is signalled for the operation:

```
CONVERSION      ENDFILE      ENDPAGE      KEY
NAME            RECORD       TRANSMIT     UNDEFINEDFILE
```

When one of these conditions is signalled, the file-name of the file on which PL/I is operating is placed on the stack associated with the ONFILE function. For example, consider the statement:

```
GET FILE(ALPHA) LIST(X);
```

If the hardware fails to correctly transmit the value of X, the TRANSMIT condition is signalled and the value of the function is:

```
ONFILE = "ALPHA"
```

Observe that the value of ONFILE is not a file value; it is a character string that is the identifier that designates the file value. When none of the conditions listed above is being handled, the value of the function is

```
ONFILE = ""      (the null string)
```

SECTION X

VALUE ASSIGNMENT

The assignment statement sets the value of a variable. This appears to be a simple action; however, two complications arise. First, the assigned value can have a different storage type from the target variable, and therefore an assignment statement sometimes requires a complicated conversion from one storage type to another. A thorough understanding of the rules given earlier, in the section on "Conversion of Values" is required. Second, the order in which the actions of assignment are performed can, in certain special cases, affect the outcome of the assignment. These problems are discussed later in this section.

The description of assignment statements in this section begins with preliminary examples; these illustrate the rules in an informal way. Next, the form and interpretation of the assignment statement is defined in detail. Finally, the pseudo variables, which are special constructs associated with value assignment, are defined.

PRELIMINARY EXAMPLES OF ASSIGNMENT STATEMENTS

The following examples are given to illustrate the considerable variety of ways in which assignment statements can be used. Examples are given for each of the major data types: arithmetic, string, address, area, and array.

Arithmetic Assignment Statements

Most of the assignment statements in a typical program are short and simple. For example, consider:

```
I = M+1;
```

where both of the variable names are declared FIXED. This statement evaluates the right-hand-side expression M+1 and obtains an 18-bit integer. That value is converted to a 17-bit integer for assignment, and, if the high-order bit was one, the SIZE condition occurs. Finally, the converted value replaces the previous value of the target variable, I.

An assignment statement can have a large right-hand-side expression and still be conceptually simple. For example, consider:

```
Z = (X-1)**2 - (A+3*B)*(X-1) + (A-3*(B/C));
```

where all the variable names are declared FLOAT. This statement is programmed entirely in floating-point binary and does not require any conversion operations. It is typical of engineering and scientific applications.

String Assignment Statements

The effect of a string assignment depends on whether or not the target variable is VARYING. For example, consider:

```
S1 = "ABC";
```

where S1 is declared CHAR(5) NONVARYING. This statement sets S1 to "ABCbb" (adding two blanks to this string) because that is the result of conversion to a CHAR(5) NONVARYING target. In contrast, consider:

```
S2 = "ABC";
```

where S2 is declared CHAR(5) VARYING. This statement sets S2 to "ABC". If the assigned value is longer than the maximum size of the target, the STRINGSIZE condition occurs, as described earlier, in the section on "Value Conversion".

A construct called a pseudo-variable can be used as the target of an assignment. A frequently used pseudo-variable is SUBSTR. It allows a portion of a string variable to be changed. For example, consider:

```
SUBSTR(S,3,2) = "XX";
```

where S is declared CHAR(8) NONVARYING. This statement sets the third and fourth characters of S to "XX" and leaves the other characters unchanged.

The major types of computational values are arithmetic, character string, and bit string. A conversion between major types is allowed, but not recommended, in GCOS PL/I. For example, consider:

```
X = "5";
```

where X is declared DEC(6,2). This statement assigns the value 0005.00 to X (which is correct) in both GCOS and Standard PL/I; but in GCOS PL/I, the compiler marks the statement with a warning. This matter is discussed earlier, under "Conversion Operations" in the section on "Operations".

Address Assignment Statements

The assignment of address values is an advanced feature of PL/I. It is especially important in list-processing. For example, consider:

```
P = P->CELL.NEXT;
```

where P is declared POINTER and CELL is a based structure whose member NEXT is declared POINTER. A statement of this form can be used to advance from one element of a list to the next element provided, of course, that the list structure is suitably defined.

Area Assignment Statements

An assignment statement can be used to copy the contents of one area variable into another. The effect is to copy the given area exactly as it appears, without any changes in the offsets or the current extent of the area. Consider the statement:

```
A1 = A2;
```

where A1 and A2 are declared AREA(100) and AREA(200), respectively. If the current extent of the AREA value of A2 is greater than 100 (the maximum value of the AREA variable A1), then the AREA condition occurs, as described earlier, in the section on "Storage Management".

Aggregate Assignment Statements

Aggregate values can be assigned to aggregate variables. For example, suppose two aggregate variables are declared as follows:

```
DCL 01 A(2),
       02 X1 FLOAT,
       02 X2 FIXED;
DCL 01 B(3),
       02 ALPHA FIXED,
       02 BETA FIXED;
```

Then the following assignment statement can be used:

```
A = B(1);
```

Observe that this statement requires the conversion of the value of the right-hand-side expression from the storage type

```
01, 02 FIXED, 02 FIXED
```

to the storage type

```
01 DIM(2), 02 FLOAT, 02 FIXED
```

The order in which the four scalar values are assigned to A, or the order in which conversions are performed, is not defined in PL/I.

THE FORM OF ASSIGNMENT STATEMENTS

The assignment statement has the following form:

t,... = *e*;

where *t* is a *target* and *e* is the *right-hand-side expression*. The form '*t*,...' indicates that the statement can contain either a single target or a sequence of targets separated by commas. A *target* is either a variable reference or a pseudo-variable reference, and the right-hand-side expression is any expression. The pseudo-variables are built into PL/I, and they are:

```
REAL(ref)
IMAG(ref)
SUBSTR(ref,e1,e2)
STRING(ref)
UNSPEC(ref)
PAGENO(ref)
ONCHAR()
ONSOURCE()
```

where *ref* is a variable reference appropriate to the pseudo-variable and *e1* and *e2* are expressions. The third argument, *e2*, of the SUBSTR function can be omitted. The pseudo-variables are defined later in this section.

Targets

In most cases, the assignment statement has a single target, and that target is a variable reference. Thus:

```
X = (A+B**N)/N;

A(I-F(Z),PHI+2) = 0;

NAME(I).LAST = "JONES";

G(ALPHA)->TAB(K+3,M) = BETA**3;
```

The statements just given illustrate the use of all four kinds of variable reference as target: simple, subscripted, structure-qualified, and locator-qualified. Examples of assignments with pseudo-variables as targets are:

```
SUBSTR(PLACE,I-3,4) = "QQQQ";
REAL(Z) = 2.8934E0;
```

Examples of assignment statements with multiple targets are:

```
K1,K2,K3 = 0;
GAMMA, G(ALPHA)->TAB(J,RHO) = PHI;
MARK, SUBSTR(PART,4,3) = "MXT";
```

THE INTERPRETATION OF ASSIGNMENT STATEMENTS

An assignment statement is interpreted as follows:

1. Evaluate Right-Hand-Side Expression. Evaluate the right-hand-side expression and save its value.

2. Convert Value and Store Through Target. Process each target one by one, starting from the leftmost and (if there is more than one target) proceeding to the right. For each target, convert a copy of the value of the right-hand-side expression to the storage type of the target and then assign the converted value to the target.

The assignment of the converted value depends on the nature of the target, as follows:

- If the target is a variable reference, then the previous value of the designated variable is replaced by the assigned value. Since the assigned value has been converted to the same storage type as the target variable, the value fits exactly into the target variable.

- If the target is a pseudo-variable, a part of the previous value of a named variable is replaced or all of a built-in variable is replaced. The effect of assignment to a given pseudo-variable is given in the definition of that pseudo-variable, later in this section.

Special Restrictions

Assignment statements must satisfy several rather special restrictions. These restrictions, together with some explanation, are given here.

OVERLAPPING STRING TARGETS

Consider the following assignment statement:

```
S = T;
```

where both variables are declared CHAR(500). The GCOS implementation of PL/I copies a string value directly from the storage for T into the storage for S without using any intermediate storage. This is an efficient implementation; however, it can be followed only if certain troublesome cases, called overlapping string targets, are excluded from the language. The restriction given here excludes those cases.

The restriction on overlapping string targets requires the definition of the special string target. A given target is a special string target if all of the following statements are true:

- The given target is a variable reference or a reference to the SUBSTR pseudo-variable.
- The given target occurs in a statement whose right-hand-side expression is either a scalar string reference or a reference to the SUBSTR built-in function. In the latter case, the first argument of the function is a scalar string variable reference.
- The string type (CHARACTER or BIT) of the target and the right-hand-side expression must be the same, so that no conversion is necessary.

When a special string target designates all or part of the storage that is designated by the right-hand-side expression of the same statement, it is an overlapping special string target. Such a target is invalid.

This restriction is designed to permit the efficient execution of assignment statements that are of a relatively simple form. The reasoning behind the restriction is as follows:

1. There are some efficient methods for executing an assignment statement that has a special string target, as already noted.
2. The efficient methods sometimes produce invalid results when they are applied to overlapping special string targets.
3. It is not always possible for the PL/I compiler to distinguish between a special string target that is overlapping and one that is not.
4. Therefore, in order to permit the use of the efficient methods, the overlapping special string targets are excluded from PL/I.

As an example, consider the program:

```
P1:  PROC;
     DCL S CHAR(100) VAR INIT("ABCDEFGHIJ");
     DCL T CHAR(100) VAR INIT("0123456789");
     SUBSTR(S,2,9) = SUBSTR(T,1,9);
     ...
     END;
```

The assignment statement has a special string target that is not overlapping. It can be efficiently executed by assigning the first character of T to the second character position of S, the second character of T to the third character position of S, and so on. The result is "A012345678", which is correct. Suppose, however, that the following assignment is used instead:

```
SUBSTR(T,2,9) = SUBSTR(T,1,9);
```

This assignment statement has an overlapping special string target and is invalid. If the statement were executed by the efficient method just described, the result could be "0012245668" instead of the presumably expected "0012345678". Therefore, to accomplish the indicated assignment, the programmer must write:

```
T1 = SUBSTR(T,1,9);
SUBSTR(T,2,9) = T1;
```

where T1 is a suitably declared string variable name.

AREA ASSIGNMENTS

The application of an assignment statement to AREA values is restricted to a statement of the following form:

> Each of the targets is a reference to a scalar AREA variable and the right-hand-side expression is a reference to a scalar AREA variable or function.

Thus an AREA value cannot be assigned as part of an aggregate value.

The Order of Interpretation

A program must not depend on the order of the steps of the interpretation unless that order is explicitly stated in the definition of the language. Programs that depend on additional assumptions about ordering are invalid. The assignment statement is a construct in which this rule is especially important.

When an assignment statement assigns a scalar value to a single target, and when the right-hand-side expression does not invoke a function that has side effects, then no problems of ordering can arise. Most assignment statements are of this convenient kind; however, those that are not must be given special attention.

Each expression in an assignment statement is evaluated according to the ordering rules for expressions, as given earlier in the section on "Expressions". In addition, the interpretation of an assignment statement is subject to the following ordering rules:

- The right-hand-side expression is evaluated before any value is assigned to a target.
- When an assignment statement has more than one target, the assignment of a value to a given target occurs before any expression (such as a subscript) in a subsequent target is evaluated.

Aside from these rules, the order of interpretation of an assignment statement is undefined.

The first of the ordering rules just given states that the right-hand expression is evaluated before a value is assigned to any target. That ordering can be important when an array value is assigned. Consider, for example, the statement:

```
A = A(2)*B;
```

where both A and B are declared DIM(3) FLOAT. Except for the ordering rule just given, one might suppose that an equivalent to this statement would be:

```
A(1) = A(2)*B(1);
A(2) = A(2)*B(2);
A(3) = A(2)*B(3);
```

This interpretation is not only incorrect but also undesirable, since it uses the old value of A(2) as the multiplier for the first two elements of B and the new value for the last. Because of the ordering requirement, the correct interpretation uses the old value of A(2) as the multiplier for all elements of B.

The definition of the assignment statement does not place a restriction on the time at which the location of a target variable begins. That is, although the assignment of a value to the target cannot occur until the evaluation of the right-hand expression is complete, there is no such restriction on the location of the target variable. In this connection, consider the statement:

```
A(I) = F(X);
```

where F is the following user-defined procedure:

```
F:   PROC(W) RETURNS(FLOAT);
       DCL W FLOAT;
       ...
       I = I+1;
       ...
       END;
```

The location of the target in the given assignment statement depends on whether the subscript in A(I) is evaluated before or after the assignment I=I+1 is executed. Therefore, the given assignment statement is ambiguous and is invalid.

The two examples just given are typical, but they do not exhaust the set of interesting examples that could be given on the subject of ordering. The designers of PL/I specified ordering where they believed it was called for by common sense and conventional notation; and they left it out where it would have been an arbitrary rule. When the programmer is in doubt, he should assume that the ordering is unspecified; and then he should take some measure to provide a positive ordering, such as breaking a statement into a sequence of statements.

PSEUDO-VARIABLES

Some of the built-in functions can be used as targets; in that context, they are called pseudo-variables, not built-in function references. A pseudo-variable can be defined, for valid arguments, in terms of its corresponding built-in function reference. Consider the assignment statement

```
REAL(Z) = REAL(Z);
```

Provided that Z is a valid argument for the REAL pseudo-variable and the REAL built-in function, this statement changes nothing. That is, the right-hand expression yields the real part of Z, and then the pseudo-variable puts that same value back as the real part of Z. All of the pseudo-variables behave in a similar manner.

A pseudo-variable can be used in only three contexts: as a target in an assignment statement (described in this section), as the target in a DO statement (as described later, in the section on "Program Flow"), and as a target in a list-directed or edit-directed GET statement (as described later, in the section on "Stream Input/Output"). In each of these cases, a value is assigned to the pseudo-variable and the interpretation of the pseudo-variable then causes a value to be assigned to storage.

A pseudo-variable name should be declared BUILTIN. Observe that a given name can be used as both a pseudo-variable name and a built-in function name in the same block; the context of each reference determines whether it is a pseudo-variable or a built-in function reference.

A complete and independent definition for each pseudo-variable is given in what follows. Each definition gives the storage type to which the assigned value is converted, and then describes what the pseudo-variable does with that value.

The REAL and IMAG Pseudo-Variables

As pseudo-variables, the references have the same forms as the built-in function references, namely:

```
REAL(Z)
IMAG(Z)
```

The value assigned to the pseudo-variable is converted to the storage type of Z except that the mode is REAL; then the converted value is assigned to the real part or imaginary part, respectively, of Z. For example, suppose the following declaration applies:

```
DCL ALPHA COMPLEX FLOAT DEC(5);
```

and suppose ALPHA has the value:

```
+5.0000E0-7.5000E01
```

Then the assignment statement

```
REAL(ALPHA) = -8.92;
```

sets the given value to -8.9200E0-7.5000E0I, while the assignment statement:

```
IMAG(ALPHA) = 0;
```

sets the given value to +5.0000E0+0.0000E0I.

The SUBSTR Function as a Pseudo-Variable

As a pseudo-variable, the reference has the same forms as the built-in function reference, namely:

```
SUBSTR(S,I,J)          SUBSTR(S,I)
```

where S must be a string variable reference and I and J must be expressions whose values can be converted to 24-bit integers. The arguments can be aggregates; however, if I or J is an aggregate, then its aggregate type must be suitable for conversion to the aggregate type of S.

Consider, first, the interpretation of the pseudo-variable when all arguments are scalars. The data type of the pseudo-variable has the string type of S (CHARACTER or BIT), but has maximum length J and the attribute NONVARYING. For example, consider:

```
SUBSTR(ALPHA,3,5) = "ABC";
```

where ALPHA is declared CHAR(10) VARYING. Here the data type of the target is CHAR(5) NONVARYING. The converted value, "ABCØØ", replaces that substring of ALPHA that begins with the third character of ALPHA and is five characters long. If the assigned value had been more than five characters long, the STRINGSIZE condition would have occurred.

There are restrictions on the pseudo-variable. First, J must be zero or positive and I and J must specify a substring that lies entirely within the limits of the current value of the variable designated by S. Thus in the example given above, the current value of ALPHA must be at least seven characters long. When this restriction is not met, the STRINGRANGE condition occurs. Second, if S is a VARYING string, a value must be assigned to it before it appears in a SUBSTR pseudo-variable; otherwise, the current length of S would be undefined.

If any of the arguments of the pseudo-variable are aggregates, they are all converted to the aggregate type of S, and this aggregate type becomes the aggregate type of the pseudo-variable. The value of the right-hand expression is, in turn, promoted to this aggregate type before assignment. After conversion, the independent components are individually assigned through the SUBSTR pseudo-variable according to the rules for scalars.

The STRING Pseudo-Variable

As a pseudo-variable, the reference has the same form as the built-in function reference, namely:

```
STRING(U)
```

where U must designate one of the following:

- A scalar string variable that is NONVARYING and ALIGNED.
- An aggregate variable whose components are all string variables of one type (either CHARACTER or BIT, but not a mixture of CHARACTER and BIT) and are all NONVARYING and UNALIGNED.

Such a variable is represented in storage as an uninterrupted sequence of characters or bits in a way that is independent of its aggregate type. Suppose the total number of characters or bits accommodated by the variable designated by U is m. Then the pseudo-variable is interpreted as follows:

- First, the value assigned to the pseudo-variable is converted to BIT(m) or CHAR(m), depending on the type of U.
- Second, if U is a scalar, the converted value is assigned to it directly; otherwise, the converted value is assigned in such a way that the concatenation of the components of the aggregate variable U are identical to the given converted value.

For example, consider the use of this pseudo-variable in the following program:

```
P:   PROC;
        DCL 01 PART UNALIGNED,
               02 CODE,
                  03 SERIAL CHAR(6),
                  03 TYPE CHAR(2),
               02 DESCRIP CHAR(10);
        ...
        STRING(PART) = "310-A6XXSIDE";
        STRING(CODE) = "680-A3";
        ...
        END;
```

The effect of the two assignment statements on the value of PART is as follows:

	first assignment	second assignment
PART.CODE.SERIAL	"310-A6"	"680-A3"
PART.CODE.TYPE	"XX"	"ØØ"
PART.DESCRIP	"SIDEØØØØØØ"	"SIDEØØØØØØ"

The UNSPEC Pseudo-Variable

As a pseudo-variable, the reference has the same form as the built-in function reference, namely:

```
UNSPEC(U)
```

Suppose that the result of the built-in function reference UNSPEC(U) would yield a bit string of length m. (The interpretation of that built-in function reference is given under "Conversion Operations" in the section on "Operations".) The pseudo-variable is interpreted as follows:

- First, the value assigned to the pseudo-variable is converted to BIT(m).
- Second, the converted value is assigned to U in a way that the function reference UNSPEC(U) yields the given converted value.

A key point is that, for any variable reference X, the assignment statement:

```
UNSPEC(X) = UNSPEC(X);
```

leaves the value of X unchanged. It then follows that the statements:

```
S = UNSPEC(X);
X =  ... ;
UNSPEC(X) = S;
```

save the value of X in S, change the value of X, and then restore the value of X.

The UNSPEC pseudo-variable is a way of interpreting the contents of raw storage (a sequence of bits) as a PL/I value. Together with the UNSPEC built-in function, it constitutes an escape from the machine-independence of PL/I and allows direct access to the storage of values in GCOS words. Suppose, for example, it is necessary to write a special PL/I procedure for converting a FIXED value to a FLOAT value. In order to write such a procedure, it might be necessary to prepare the mantissa and the exponent separately and then combine them into a single FLOAT value by means of the UNSPEC pseudo-variable.

The PAGENO Pseudo-Variable

As a pseudo-variable, the reference has the same form as the built-in function reference, namely:

```
PAGENO(F)
```

where F must yield a scalar file value that has the PRINT attribute. The value assigned to the pseudo-variable is converted to a 35-bit integer. For example, suppose ALPHA is declared as a file-name constant. Then the assignment statement:

```
PAGENO(ALPHA) = 100;
```

sets the page number counter for file ALPHA to 100.

The ONCHAR and ONSOURCE Pseudo-Variables

As pseudo-variables, the references have the same forms as the built-in function references, namely:

ONSOURCE	ONSOURCE()
ONCHAR	ONCHAR()

The value assigned to ONSOURCE is converted (if necessary) to a character-string value and then replaces the value currently at the top of the stack that is associated with the ONSOURCE built-in function. The value assigned to ONCHAR is converted (if necessary) to a CHAR(1) value and then replaces the conversion character (as defined below) in the character-string value currently at the top of the stack associated with the ONSOURCE built-in function.

The ONSOURCE and ONCHAR pseudo-variables are associated with the CONVERSION condition. The CONVERSION condition occurs when an attempt is made to convert a character-string value or pictured value to an arithmetic or bit-string value and the attempt fails because the given string has the wrong form. The conversion character is the leftmost character that must be changed as part of the correction of the given character-string value. A more detailed discussion of these matters is given earlier, under "The ON Condition Functions" in the section on "Operations".

As an example of the use of ONCHAR as both a pseudo-variable and a built-in function, consider the following program:

```
P:   PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL X FLOAT;
        DCL CONV COND;
        DCL ONCHAR BUILTIN;
        ON CONV
             BEGIN
             PUT SKIP LIST("ERROR IN INPUT");
             IF ONCHAR= "I" THEN ONCHAR = "1";
                  ELSE IF ONCHAR= "O" THEN ONCHAR = "O";
                  ELSE SIGNAL ERROR;
             END;
        GET LIST(X);
        PUT SKIP LIST(X**2);
        END;
```

This program computes the square of a given number. The ON statement provides the program with a primitive form of error recovery: when the input value has either of the letters I or O, the program prints a warning message and assumes that the corresponding digit was intended.

Suppose that the following input is supplied to this program:

```
I23EI
```

The conversion of this input (which is a character-string value) for assignment to X (which requires a FLOAT value) proceeds as follows:

1. The first attempt at conversion fails. The first character, which is the letter I, is the conversion character. The CONVERSION condition is signalled. The ON unit prints a warning and replaces the conversion character with the digit 1. Then the PL/I processor again attempts the conversion.

2. The second attempt also fails. The fifth character is also the letter I, and is handled in the same way as the first I.

3. The third attempt at conversion succeeds because the given string is "123E1".

Although this example shows how the ONCHAR pseudo-variable works, it is a simple example of recovery. A more sophisticated recovery procedure would use the ONSOURCE built-in function to obtain the entire incorrect string, would analyze that string and make appropriate changes, and would use the ONSOURCE pseudo-variable to replace the entire incorrect string with the corrected version.

SECTION XI

PROGRAM FLOW

As a program is executed, the PL/I interpreter passes from one statement to another; this part of program execution is the flow of control. There are seven kinds of flow of control in PL/I, as follows:

- Sequential flow is the execution of statements in the order in which they appear in the program. This kind of flow of control is used wherever some other kind is not explicitly called for.

- Conditional flow uses a test of current data values to determine whether or not a statement is executed. The IF statement is used for this purpose. A set of IF statements can be nested, one within the other, so it is possible to program a complicated case analysis using only IF statements.

- Iterative flow uses various conventions to execute a group of statements repeatedly. The DO statement is used to control the iteration. An index can be associated with the iteration.

- Transfer of control sends control to a specified statement. The GOTO statement is used for this purpose. Because PL/I has both arrays of LABEL constants and unlimited LABEL variables, a rather general computation can be used to obtain the destination of the transfer.

- Block execution applies to a BEGIN block. A BEGIN block is used to declare variables in a restricted scope.

- Procedure invocation executes a block of statements as a closed subroutine. The CALL statement or the function reference is used for procedure invocation. Provision is made for the transmission of arguments by either value or address, and procedures can be invoked recursively.

- Condition handling is used to process exceptional conditions that can occur during program execution, such as a division by zero or a transmission failure during input. The ON, REVERT, and SIGNAL statements are used for this purpose.

The first five kinds of flow of control are described in this section. The last two kinds require separate treatment and are described in the subsequent sections, "Procedure Invocation" and "Condition Handling".

SEQUENTIAL EXECUTION

When execution is sequential, statements are executed in the order in which they appear. However, when control reaches the end of a procedure, there is no "next statement" in the sequence; so the PL/I interpreter acts as if the next statement is

```
RETURN;
```

and thus returns to the point at which the procedure was invoked.

The following constructs cause no action when they are encountered during sequential execution:

```
PROCEDURE blocks
FORMAT statements
DECLARE and DEFAULT statements
```

Some of these constructs (PROCEDURE blocks and FORMAT statements) cause action only when they are invoked by a proper form of reference. The remaining constructs never cause an action; they are present only to supply declaration information.

As an example of sequential execution, consider the following program:

```
P:   PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL (A,B,C) FLOAT;
SQ:     PROC(X,Y) RETURNS(FLOAT);
        DCL (X,Y,Z) FLOAT;
        Z = (X**2 + 5*Y)/X;
        RETURN(Z);
        END;
        GET LIST(A,B);
        C = 2*SQ(A,B);
        PUT SKIP LIST(A,B,C);
        END;
```

Execution of the program is summarized as follows:

1. The PROCEDURE statement, the two DECLARE statements, and the internal PROCEDURE block are executed in sequence. The sequential execution of these constructs causes no action.

2. The input statement is executed and gets values for A and B from the input stream.

3. The assignment statement is executed. As part of its execution, it invokes the procedure SQ; during that invocation, the statements in that procedure are executed sequentially.

4. The output statement is executed and prints the results.

5. Because the end of the external procedure has been reached, PL/I assumes a RETURN statement and returns to the command that invoked the external procedure.

THE IF STATEMENT

An IF statement has one of the following forms:

IF t THEN c1 ELSE c2

IF t THEN c1

where t is the test and c1 and c2 are the consequences. The test is usually a relational expression, whose value is "1"B or "0"B (that is, true or false). Each consequence is usually a single statement.

An IF statement that has an ELSE clause (the first form), is interpreted as follows:

1. Evaluate the test.
2. If the test value is true, then execute the first consequence; otherwise execute the second consequence.

Consider, for example:

```
IF X >= 2*B
     THEN Z = SQRT(X-2*B);
     ELSE CALL ERR3(X);
```

Here, the test is the relational expression X>=2*B, the first consequence is the assignment ''''Z=SQRT(X-2*B);'''', and the second consequence is the statement ''''CALL ERR3(X);''''. This IF statement executes the assignment statement if the argument of the square root is non-negative; otherwise, it calls an error routine.

An IF statement without an ELSE clause takes no action when the value of the test is false. Consider, for example:

```
IF I<0 THEN I=0;
```

Depending on whether I is negative or not, this statement sets I to zero or takes no further action.

An IF statement without a THEN clause is not part of the language. However, a specification that requires an omitted THEN clause can easily be converted to one that requires an omitted ELSE clause; and the resulting uniformity is beneficial. For example, consider the specification:

> If X=0 is true, then do nothing; otherwise, execute the assignment statement Y=1/X;.

This specification can be converted to a suitable form by negating the test and exchanging the consequences. The result is:

> If X^= 0 is true, then execute the assignment statement 'Y=1/X;'; otherwise, do nothing.

The specification can now be written in PL/I, as follows:

```
IF X^= 0 THEN Y = 1/X;
```

The Test in an IF Statement

The test in an IF statement can be any expression that yields a scalar bit-string value. Usually, the bit-string value is of length one, but any length is allowed. If any bit of the test value is one, then the test is true; otherwise, the test is false.

The test can specify a lengthy computation; for example:

```
IF ABS(X) = 0
        & SQRT(U**2+V**2)<F(R+1)
        & (SW1=5 ! SW1=9)
     THEN CALL R1(Z);
     ELSE CALL R2(Z);
```

Here the test is a complicated bit-string expression; furthermore, the test contains the reference F(R+1), which (it is assumed) is a reference to a user-defined function and which can require considerable computation in itself. When the computation is complete, however, the final result is simply "1"B or "0"B.

Although the definition of the test in an IF statement permits any expression that yields a bit-string value, an expression that does not yield a BIT(1) NONVARYING value should be avoided. A multiple-bit test can give unexpected results; for example, consider the following statements:

```
IF B THEN CALL R1; ELSE CALL R2;

IF ^B THEN CALL R2; ELSE CALL R1;
```

It is natural to expect these statements be equivalent to one another; however, they are equivalent only if the value of B is a string of zeros only or of ones only, and that can be assured only if B is BIT(1) NONVARYING. Suppose, for example, that B is declared BIT(2) and the assignment statement

```
B = "01"B;
```

has been executed. In the first statement, the test is true because it contains a one bit and therefore R1 is called. In the second statement, the value of the test is "10"B and so it is once again true and R2 is called. Thus the statements are not equivalent when B is declared BIT(2).

The Consequences in an IF Statement

Each consequence in an IF statement must be an executable unit. The constructs that are executable units are:

- The DO group; that is, a DO statement followed by a sequence of statements followed by an END statement
- The BEGIN block; that is, a BEGIN statement followed by a sequence of statements followed by an END statement

- The independent statement; that is, one of the following kinds of statement:

storage management:	ALLOCATE and FREE
assignment:	the assignment statement
flow of control:	IF, GOTO, and the null statement
procedure invocation:	CALL and RETURN
condition handling:	ON, REVERT, and SIGNAL
input/output:	OPEN and CLOSE
stream input/output:	GET and PUT
record input/output:	READ, WRITE, DELETE, REWRITE and LOCATE

The definition just given allows the use of nearly any group, block, or statement as a consequence. Those not allowed are:

- Constructs that cause no action when encountered through sequential flow of control; for example, a PROCEDURE block, a FORMAT statement, or a DECLARE statement.

- Statements that are not complete in themselves but must be part of a larger construct; for example, the DO statement, which must be part of a DO group, or an ENTRY statement, which must be part of a PROCEDURE block.

The broad definition of the consequence means that the use of a GOTO statement to transfer around statements is unnecessary. For example, instead of writing:

```
        IF X=0 & Y<2 THEN GOTO L3;
        DO I = 1 to N;
             A(I) = B(I)*C(I);
             Q(I) = 0;
             END;
L3:     ...
```

one should write:

```
        IF^ (X=0 & Y<2) THEN
             DO I = 1 TO N;
                  A(I) = B(I)*C(I);
                  Q(I) = 0;
                  END;
        ...
```

This form avoids the use of a label prefix; furthermore, its syntactic structure corresponds to the logical structure of the computation.

Two of the constructs that can be used as consequences are of particular significance. They are the non-iterative DO group and the IF statement itself; and they are given special attention in the following paragraphs.

THE NON-ITERATIVE DO GROUP AS A CONSEQUENCE

A non-iterative DO statement is one that has no WHILE option or index. A non-iterative DO group begins with a non-iterative DO statement, and its sole purpose in the language is to gather together two or more statements so that they can be treated as a single consequence in an IF statement. Consider, first, the following two program fragments:

```
IF X=0 THEN Z=1; CALL Q(ALPHA);

IF X=0 THEN DO; Z=1; CALL Q(ALPHA); END;
```

These program fragments have different meanings. The first fragment is an IF statement followed by a CALL statement; and the two statements are executed independently, one after another. The second fragment is an IF statement whose consequence is a non-iterative DO group; and the entire DO group is executed or not depending on whether the value of X is zero or not.

Another example of the use of a non-iterative DO group is:

```
IF A = B
   THEN DO;
        ALPHA = F(3,A-2) - F(3,2*B);
        BETA  = G(0);
        GAMMA = 4;
        END;
   ELSE DO;
        ALPHA = F(3*(A-B)+1,2*(A-1)-B) - F(3,2*B);
        BETA  = G((A-B)/2);
        GAMMA = H(2*(A-B)+4);
        END;
```

Provided such statements are laid out on the page in a clear and uniform way, they are easy to read and understand.

THE IF STATEMENT AS A CONSEQUENCE

As a case of particular interest, the consequence of an IF statement can, itself, be an IF statement; that is, IF statements can be nested. There is no limit to the depth of this nesting, and it is not uncommon for nesting to have three levels.

Suppose that the following case analysis is given as part of the specification of a program:

Test	Expression for Z
X>0	ATAN(Y/X)
X=0 & Y>0	PI/2
X=0 & Y=0	(undefined)
X=0 & Y<0	-PI/2
X<0 & Y>=0	ATAN(Y/X)+PI
X<0 & Y<0	ATAN(Y/X)-PI

This table gives five different expressions for Z, depending on the values of X and Y; it also shows the range of values for which Z is not defined. The case analysis can be programmed <u>without</u> nesting as follows:

```
IF X>0 THEN Z = ATAN(Y/X);
IF X=0 & Y>0 THEN Z = PI/2;
IF X=0 & Y=0 THEN SIGNAL ERROR;
IF X=0 & Y<0 THEN Z = -PI/2;
IF X<0 & Y>=0 THEN Z = ATAN(Y/X)+PI;
IF X<0 & Y<0 THEN Z = ATAN(Y/X)-PI;
```

This version is readable but not efficient. There are 11 relational expressions in the program fragment, and every one of them is evaluated every time the fragment is executed.

An efficient programming of the case analysis given in the table is:

```
IF X>0
     THEN Z = ATAN(Y/X);
     ELSE IF X=0
          THEN IF Y>0
               THEN Z = PI/2;
               ELSE Z = -PI/2;
     ELSE IF Y>=0
          THEN Z = ATAN(Y/X)+PI;
          ELSE Z = ATAN(Y/X)-PI;
```

There are five relational expressions in this version (instead of 11); and, if the cases occurred with equal frequency, an average execution of the program fragment would require the evaluation of three relational expressions (instead of 11).

The Dangling ELSE Clause

The omission of an ELSE clause in an IF statement that is part of a nest of IF statements can produce confusion. Consider the following statement:

```
IF X<0 THEN IF A<0 THEN Y=A/X; ELSE Y=0;
```

Does the ELSE clause go with the entire statement (so that it is executed when X<0 is false) or does it go with the nested IF statement (so that it is executed when X<0 is true and A<0 is false). Because the answer to this question is not obvious, ELSE Y=0; is called a "dangling ELSE clause".

The rules of PL/I supply the answer: an ELSE clause is always associated with the <u>smallest</u> possible IF statement. Therefore, the ELSE clause in the example goes with the nested IF statement, and the correct layout for the statement is:

```
IF X<0
     THEN IF A<0
          THEN Y=A/X;
          ELSE Y=0;
```

The problem with the dangling ELSE arises when a programmer wants the ELSE clause to be associated with the entire statement. Then either of the following can occur:

1. The programmer forgets the rules and writes:

```
IF X<0
     THEN IF A<0
          THEN Y=A/X;
     ELSE Y=0;
```

   This formatting is wrong, but, unfortunately, it looks reasonable.

2. The programmer remembers the rules and writes:

```
IF X<0
     THEN DO;
          IF A<0 THEN Y=A/X;
          END;
     ELSE Y=0;
```

   This version is correct but it requires the introduction of a DO group.

A general solution to this problem is to use an ELSE clause with <u>every</u> IF statement in a nest of IF statements. When there is no action for the ELSE clause to perform, a <u>null statement</u> can be used as the consequence. A null statement is the single character ';' and its execution causes no action.

With this approach in mind, the example can be written to make both interpretations clear. To obtain the first interpretation, write:

```
IF X<0
     THEN IF A<0
          THEN Y=A/X;
          ELSE Y=0;
     ELSE;
```

To obtain the second interpretation, write:

```
IF X<0
     THEN IF A<0
          THEN Y=A/X;
          ELSE;
     ELSE Y=0;
```

<u>THE DO GROUP</u>

There are three kinds of DO group:

iterative DO without index
iterative DO with index
non-iterative DO

A DO group gathers together a sequence of statements for execution as a single unit. This gathering together is the only purpose of a non-iterative DO group. An iterative DO group does more: it executes the statements that are gathered together repeatedly, and is used to program loops. Every DO group, iterative or not, eliminates at least one GOTO statement from a program, and the result, in most cases, is an important contribution to the clarity of the program.

The general form of a DO group is

```
DO ... ;
     ss
     END;
```

where ss is the body of the group. The body is a sequence of any number of statements. The notation 'DO ...;' is used to indicate that, at this point in the discussion, the details of the DO statement are being left out; details are given later, as each kind of DO group is described. Although label prefixes are not shown above, the END statement can be preceded by one or more label prefixes, and can be the destination of a transfer of control.

A transfer of control from a GOTO statement that is outside an iterative DO group to a statement that is inside the group is not valid. The only way to enter an iterative DO group is by flowing into or transferring to the DO statement at the beginning of the group.

The Iterative DO without Index

An iterative DO group without an index has the form

```
DO WHILE(t);
     ss
     END;
```

where t is the test and ss is the body of the DO group. The test is defined in the same way as the test in an IF statement; that is, it must yield a scalar, bit-string value. The test is true if at least one bit is one and is false otherwise. The use of a test bit-string with length other than one is not recommended. When this recommendation is followed, the value of the test is "1"B or "0"B.

The iterative DO group without an index executes the body of the group repeatedly while the value of the test is true. The detailed interpretation is:

1. Evaluate the test.
2. If the test value is false, execution is complete. Otherwise,
3. Execute the statements of the body of the group; that is, start with the first statement of the body and continue until the END statement is executed.
4. Go to Step 1.

The iterated DO group without index can be used to write the most primitive kind of loop: one that appears to go on forever. In practice, such loops are often required. Consider, for example, the program:

```
P:   PROC;
     DCL (SYSIN,SYSPRINT) FILE;
     DCL X FLOAT;
     DO WHILE("1"B);
          GET LIST(X);
          PUT SKIP LIST(SQRT(X));
          END;
     END;
```

This program turns the computer into a calculator of square roots; it runs until the input stream is exhausted. The DO statement shown here is used wherever the programmer must design his own loop control rather than using one of the methods of control provided by PL/I.

The iterative DO group without an index is well suited to control of the computation of a mathematical approximation. Suppose a quantity, Y(X), must be computed for a given value of X. The function Y is not given as a formula; instead, the following user-defined functions are available for use in the program:

INITIAL_GUESS(X)	which gives the first approximation for Y(X) in terms of the given X
BETTER_GUESS(X,OLDY)	which gives the (i)th approximation for Y(X) in terms of X and the (i-1)th approximation, OLDY

The following statements compute successive approximations to Y until two successive approximations differ by no more than one ten-thousandth of the value of the current approximation:

```
OLDY = INITIAL_GUESS(X);
NEWY = BETTER_GUESS(X,OLDY);
DO WHILE(ABS(NEWY-OLDY) > .0001E0*ABS(NEWY));
     OLDY = NEWY;
     NEWY = BETTER_GUESS(X,OLDY);
     END;
```

When this program fragment is written without the benefit of the DO group, it is:

```
     OLDY = INITIAL_GUESS(X);
     NEWY = BETTER_GUESS(X,OLDY);
LOOP: IF ^(ABS(NEWY-OLDY) > .0001E0*ABS(NEWY))
          THEN GOTO DONE;
     OLDY = NEWY;
     NEWY = BETTER_GUESS(X,OLDY);
     GOTO LOOP;
DONE: ...
```

This version requires two GOTO statements and their accompanying label prefixes. It makes details explicit but obscures the general intent of the programmer.

The Iterative DO with Index

An iterative DO group with an index has the form:

```
DO i = cl;
     ss
     END;
```

where *i* is the *index*, *cl* is the *control list* and *ss* is the *body* of the group. The index is usually a simple reference to a scalar, but other possibilities are described later. The control list is a sequence of controls separated by commas.

Execution of an iterative DO group with an index is divided into phases. The first control in the control sequence governs the first phase, the second control governs the second phase, and so on. Consider the following DO group:

```
DO I = 1 BY 2 TO 5, 13;
     A(I) = B(I);
     END;
```

In this example, the control list consists of two controls. The first phase of execution is governed by the control 1 BY 2 TO 5 and executes the body of the group three times. The second phase is governed by the control 13 and executes the body once. The DO group is equivalent to

```
A(1) = B(1);
A(3) = B(3);
A(5) = B(5);
A(13) = B(13);
```

There are three kinds of controls, as follows:

The single-value control
The REPEAT control
The FORTRAN control

Detailed descriptions of these controls follow. The controls are quite different from one another, but they have the WHILE option in common, which allows a control phase be cut short when the test in the option is false.

THE SINGLE-VALUE CONTROL

The single-value control has the form:

e WHILE(*t*)

where *e* is an expression and *t* is the *test*. The expression must yield a value suitable for assignment to the index. The *test* is defined as in an IF statement. The WHILE(*t*) option can be omitted.

A single-value control can be used to supply an index value for a single execution of a DO group. The detailed interpretation is:

1. Execute the assignment statement

   i = *e*;

2. If the WHILE option is present, evaluate the test. If the test value is false, then exit from this phase.

3. Execute the statements of the body of the DO group; that is, start with the first statement and continue until the END statement has been executed.

4. Exit from this phase.

The DO group in the following program has three controls, each a single-value control:

```
P:    PROC;
      DCL SYSPRINT FILE;
      DCL S CHAR(20) VAR;
      DO S = "RED", "YELLOW", "BLUE";
           PUT SKIP LIST("THE COLOR IS "!!S!!".");
           END;
      END;
```

The program prints:

```
THE COLOR IS RED.
THE COLOR IS YELLOW.
THE COLOR IS BLUE.
```

THE REPEAT CONTROL

The REPEAT control has the form:

e1 REPEAT e2 WHILE(t)

where e1 and e2 are expressions and t is the test. The expressions must yield values suitable for assignment to the index. The test is defined as in the IF statement. The WHILE(t) option can be omitted.

A REPEAT control has one expression to supply the value for the first execution of the body of the group and a second expression for subsequent executions. The detailed interpretation is:

1. If this is the first execution of this step (Step 1.) in the current phase, then execute the assignment:

   i = e1;

   Otherwise, execute the assignment:

   i = e2;

2. If the WHILE option is present, evaluate the test. If the test is false, then exit from this phase.

3. Execute the statements of the body of the DO group; that is, start with the first statement.and continue until the END statement has been executed.

4. Go to Step 1.

The following program fragment prints all of the powers of two that are between 1 and 100:

```
DO I = 1 REPEAT 2*I WHILE(I<100);
     PUT LIST(I);
     END;
```

The DO group is equivalent to:

```
        I = 1;
LOOP:   IF ^(I<100) THEN GOTO DONE;
        PUT LIST(I);
        I = 2*I;
        GOTO LOOP;
DONE:   ...
```

The REPEAT control is especially useful for searching a linked list. Suppose the array that holds the list is declared as follows:

```
DCL 01 CELL(500),
       02 CODE CHAR(6),
       02 CUSTOMER CHAR(30),
       02 LINK FIXED(9);
```

Suppose that when this array is in use:

- CELL(1) is the first member of the list.
- If CELL(I) is the (k)th member of the list, then CELL(CELL.LINK(I)) is the (k+1)th member of the list.
- If CELL(I) is the last member of the list, then CELL.LINK(I) is zero.

Consider the following DO group:

```
DO I = 1 REPEAT CELL.LINK(I) WHILE(I ^= 0);
     IF CELL.CODE(I) = GIVEN_CODE THEN GOTO FOUND;
     END;
GOTO NOT_FOUND;
```

These statements search the list for the first member that has a CELL.CODE that is identical to that contained in GIVEN_CODE.

THE FORTRAN CONTROL

The FORTRAN control has one of the forms:

e1 BY e2 TO e3 WHILE(t)

e1 TO e3 BY e2 WHILE(t)

where e1 is the initialization, e2 is the increment, e3 is the limit, and t is the test. The expressions are subject to the following restrictions, which reflect the uses to which the expressions are put:

- The initialization must be suitable for assignment to the index.
- The increment must be a suitable operand for the addition operation.
- The limit must be a suitable operand for the < operator.

The test is defined as in the IF statement. The WHILE(t) option can be omitted. Either the BY e2 clause or the TO e3 clause can be omitted, but not both.

A FORTRAN control is often used to supply a sequence of index values that is an arithmetic progression; indeed, in many cases, it is used to obtain the sequence of the first n integers:

1, 2, ..., n

Although such uses are simple, the full interpretation of the control is complicated. The complexity results from the concern of the designers for efficiency; specifically, from the decision to evaluate the expressions associated with the index, the initialization, the increment, and the limit only once during a phase.

The detailed interpretation of the FORTRAN control is:

1. Prepare for execution of the loop by performing the following actions in any order:

   a. Determine the location of the storage unit designated by i (the index of the group). (In order to do this, any expressions in i, such as subscript expressions, must be evaluated). Save the location of the storage unit, and wherever i appears in these steps, use the saved location rather than re-evaluating the expressions in i.

   b. If the BY e2 clause is present, evaluate e2 and save the value. Use the saved value wherever e2 appears in these steps.

   c. If the TO e3 clause is present, evaluate e3 and save the value. Use the saved value wherever e3 appears in these steps.

2. If this is the first execution of this step (Step 2) in the current phase, then execute the statement:

   i = e1;

   Otherwise, execute one of the following:

   i = i + e2; (if the BY e2 clause is present)

   i = i + 1; (if the BY e2 clause is absent)

3. If the WHILE option is present, evaluate the test. If the test is false, then exit from this phase.

4. If the TO e3 clause is present, test the value of i against e3 as follows: If the BY e2 clause is not present and i>e3, then exit from this phase. If e2 is positive and i>e3, then exit from this phase. If e2 is negative and i<e3, then exit from this phase.

5. Execute the statements of the body of the DO group. Execute them in the normal way; do not use any of the information saved in Step 1. Start with the first statement and continue until the END statement has been executed.

6. Go to Step 2.

A simple example of a DO group with a FORTRAN control is:

```
DO X = -90 BY .25 TO 90;
     PUT SKIP LIST(X, SIND(X), COSD(X));
     END;
```

This group prints the values of the sine and cosine functions from -90 degrees to +90 degrees in increments of .25. It is equivalent to the statements:

```
        X = -90;
LOOP:   IF X>90 THEN GOTO DONE;
        PUT SKIP LIST(X, SIND(X), COSD(X));
        X = X+.25;
        GOTO LOOP;
DONE:   ...
```

A short example that illustrates the consequences of Step 1 of the interpretation is:

```
        A=0;
        B=2;
        C=10;
        K=1;
        DO A(K) = A BY B TO C;
             K = K+1
             B = -B;
             C = C-1;
             END;
```

These statements are equivalent to:

```
        DO A(1) = 0 BY 2 TO 10;
             K = K+1;
             B = -B;
             C = C-1;
             END;
```

If the current values of the control parameters were used for each execution of the loop, the interpretation of this loop would be extremely complicated; to start with, it would require a knowledge of all of the elements of the array A.

THE INDEX OF A DO GROUP

There are two special restrictions on the index of a DO group, as follows:

- The index must designate a scalar value.
- The index must not designate an AREA value.

Aside from these restrictions, the index can be any construct that is a valid target for the assignment statements that are explicitly shown in the interpretation of the DO group.

The following program illustrates the use of a pseudo-variable as the index of a DO group:

```
P:    PROC;
      DCL SYSPRINT FILE;
      DCL Z COMPLEX FLOAT;
      DO REAL(Z)^= 0 BY .25 TO 1;
           DO IMAG(Z) = 0 BY .25 TO 1;
                IF Z ^= 0 THEN
                     PUT SKIP LIST(Z,1/Z);
                END;
           END;
      END;
```

This program lists Z and 1/Z for 24 values of the complex variable Z. The program shows that one DO group can be used within another. There is no restriction on the depth of the nesting that can be used.

The Non-Iterative DO

A non-iterative DO group has the form:

```
DO;
     s1
     s2
     ...
     END;
```

where s1, s2, and so on are the body of the group. The group is executed by executing the sequence of statements s1, s2, and so on, once.

The effect of the non-iterative DO group is to gather s1, s2, and so on into a single syntactic unit. The only use for this kind of DO group is as a consequence of an IF statement, as described earlier in this section.

THE GOTO STATEMENT

A GOTO statement has the form:

```
GOTO ref;
```

where ref is any reference that yields a scalar LABEL value. When the statement is executed, the reference is evaluated and control transfers to the statement designated by the LABEL value. The statement to which control transfers is the destination of the transfer of control. If the destination is outside of the block that contains the GOTO statement, then the transfer causes control to exit from that block; block exits are described later, in the section on "Procedure Invocation". If the program is recursive, additional rules are required to determine the destination; these are given later, again under "Procedure Invocation", and they apply only to certain advanced and unusual situations.

The GOTO with a Constant Reference

When the reference in a GOTO statement is a constant reference, the destination must be a statement in the same block or in a containing block. The reference is further restricted by the fact that a LABEL constant must be either a scalar or a one-dimensional array.

A GOTO statement whose reference designates an element of a LABEL array constant is called a switch. An example of the use of a switch is:

```
          GOTO C(I+1);
C(1):     Z = X**3 + Y**3;  GOTO DONE;
C(2):     Z = X/(Y**2-X);  GOTO DONE;
C(5):     Z = 1;
DONE:     ...
```

Because it is an array subscript, the value of I+1 is converted to an integer by dropping the fractional digits. Then the switch is interpreted as follows:

TRUNC(i+1)	Action
0 or less	the SUBSCRIPTRANGE condition occurs
1	the assignment labelled C(1) is executed
2	the assignment labelled C(2) is executed
3	(undefined)
4	(undefined)
5	the assignment labelled C(5) is executed
6 or more	the SUBSCRIPTRANGE condition occurs

Observe that no condition occurs when an element is missing from within the array of labels.

The GOTO with a Non-Constant Reference

The use of a variable reference or a function reference in a GOTO statement is usually confined to advanced applications or entirely avoided.

As an example of the use of a variable in a GOTO statement, consider the procedure:

```
IGL: PROC(X,A,ERROR) RETURNS(FLOAT);
          DCL (X,A) FLOAT;
          DCL ERROR LABEL;
          DCL T FLOAT;
          T = X**2 + A**2;
          IF T<0 THEN GOTO ERROR;
          RETURN(LOG(X+T));
          END;
```

This procedure transfers to the statement designated by the variable ERROR when T is negative. The variable ERROR is a parameter, and its value is supplied by the reference to the procedure. An example of such a reference is:

```
     ALPHA = (BETA + IGL(GAMMA,2,LAB))/DELTA;
     ...
LAB: PUT SKIP LIST("IGL FAILED AT ALPHA");
     GOTO EXIT;
```

THE LOCAL ATTRIBUTE

The transfer of control performed by a GOTO statement can be either local or non-local. A local transfer is one for which both the GOTO statement and its destination are immediately contained in the same block. A non-local transfer is any other transfer. A local transfer can be executed at considerably less cost than a non-local transfer.

The PL/I interpreter can perform a local transfer efficiently only when the transfer can be recognized as such before execution. Three cases apply:

- If the reference in the GOTO statement is a constant reference, a local transfer can be recognized by examination of the program.
- If the reference is a variable reference, a local transfer can be recognized only if the variable name is declared with the LOCAL attribute.
- If the reference is a function reference, a local transfer cannot be recognized before execution.

As an example of the use of the LOCAL attribute, consider the following program fragment:

```
          DCL X LOCAL LABEL;
          ...
          IF Z=0 THEN X=LAB1; ELSE X=LAB2;
          ...
          GOTO X;
LAB1:     M = ALPHA + BETA**2; GOTO DONE;
LAB2:     M = ALPHA - BETA**2; GOTO DONE;
DONE:     ...
```

The use of the LOCAL attribute is never essential. If the LOCAL attribute were omitted from the declaration of X, these statements would still be correct; but the execution of the GOTO X; would cost more.

A RESTRICTION ON THE DESTINATION

The destination of a GOTO statement must be a statement that is immediately contained in an active block. If the reference in the GOTO statement is a constant reference, this restriction requires that the destination be in some block that contains the GOTO statement, and the compiler checks for compliance. If the reference is a variable reference or a function reference, the matter is neither so simple nor so safe.

The following program illustrates the problem:

```
P:    PROC;
      DCL X LABEL;
P1:   PROC;
            ...
            X = LAB;
            ...
LAB:        ...
            END;
P2:   PROC;
            ...
            GOTO X;
            ...
            END;
      CALL P1;
      CALL P2;
      END;
```

This program executes the procedure P1 and then the procedure P2. The first procedure assigns the label value designated by LAB to the label variable X. At the time of assignment, the statement designated by LAB is in an active block. Later, the second procedure attempts to execute 'GOTO X;'; but at that time the block that contains the statement designated by LAB is no longer active. Therefore the transfer is not valid.

There is no way to detect this invalid transfer in advance without understanding the logic of the program; therefore, the error is not detected by the compiler.

BLOCK EXECUTION

The principal purpose of a BEGIN block is to establish declarations. The recommended style of programming in GCOS PL/I is to write short PROCEDURE blocks. Since the necessary declarations can be established in these PROCEDURE blocks, a BEGIN block is rarely needed; indeed, some programmers never use them.

Control can enter a BEGIN block only by execution of the BEGIN statement that is the first statement of the block. The statement is executed either in sequence or by transfer of control to a label prefix in the statement. When the BEGIN statement is executed, it does not cause any action directly, but the entry to the BEGIN block causes the block to be activated.

Control can exit from a BEGIN block by execution of the END statement that is the last statement of the block. Alternatively, control can exit by execution of a GOTO statement whose destination lies outside the block. In either case, exit from the block causes the block to be deactivated.

The activation and deactivation of a block are described later, in the section on "Procedure Invocation".

GUIDELINES FOR FLOW OF CONTROL

When the PL/I facilities for programming flow of control are used effectively, they make the logic of a program seem simple and thus allow the programmer to concentrate on the details of the operations being performed. Two factors are important in PL/I programming: avoidance of unnecessary GOTO statements and proper use of page layout. These factors are discussed here.

Avoidance of Unnecessary GOTO Statements

A common source of unclear programming is the use of a GOTO statement where IF statements, DO groups or procedure calls could be used. The avoidance of these GOTO statements requires more work in the design of a program but reduces the work of debugging; the net result is an improvement of the program.

Some uses for the GOTO statement are necessary in PL/I because they can be avoided only by obscure tricks. The use of a switch is sometimes essential and the use of a GOTO statement to escape from a loop is common. But every use of a GOTO should be considered carefully and retained only if it fills a special need. There are not many reliable rules for good programming style; but avoidance of unnecessary GOTO statements is one such rule.

Layout Conventions

A program is arranged in an attractive layout by means of blanks, tabs, and newlines. The PL/I interpreter ignores the layout of a program, and leaves this responsibility entirely in the hands of the programmer. The programmer therefore has two tasks:

- He must choose conventions that provide for program layouts.
- He must detect his own errors in his application of the layout conventions.

There is some variation in the layout conventions used by PL/I programmers. The following rules are used for the example programs in this manual:

1. Start each statement on a new line. This keeps statements from getting lost.
2. If a statement, group, or block requires more than one line, indent every additional line relative to the first line. This makes it easy to find the end of the statement, group, or block.
3. If an IF statement with an ELSE clause requires more than one line, begin a new line for the THEN clause and a new line for the ELSE clause. Start the ELSE clause in the same column as the THEN clause. This makes it easy to match the clauses of an IF statement.
4. Put every label prefix at the left margin, even when the statement of which it is a part is indented. This makes it easy to search for a particular label prefix.

These rules have exceptions. For example, a consequence should begin on the same line as the THEN or ELSE that precedes it. If a statement is very closely related to the statement that precedes it, it can appear on the same line; this is sometimes true of a GOTO statement.

SECTION XII

PROCEDURE INVOCATION

Programs can be written quickly and accurately if the top-down structured approach to programming is used. This approach is applied to a given problem as follows:

1. Call the problem the current task.

2. Program the current task as a procedure. If the task is large and complicated, factor out a subtask; that is, select a coherent portion of the task, give it a name, and replace it by a CALL statement or function reference to a procedure that will be written later. Place the factored subtask on the list of remaining tasks. Continue to factor out subtasks in this way until the current task can be written as a simple procedure that is about one page long.

3. If there are any remaining tasks, let one of them be the current task and go to Step 2. Otherwise, the program for the given problem is complete.

Top-down structured programming depends heavily on the availability of a complete and efficient facility for writing and invoking procedures. PL/I fills that requirement well. Although the large number of procedure invocations introduced by the approach can increase the expense of executing a program, that expense is compensated for by the reduction in the cost of program development and maintenance. Further, the GCOS implementation of PL/I includes special optimizations designed to reduce the cost of procedure invocation. From all of this, it follows that procedure invocation is perhaps the most important feature of GCOS PL/I.

The subject of procedure invocation is presented here in an order that allows explanation and motivation of each topic. The discussion begins with the passing of arguments to parameters since all the features of procedure invocation depend on this operation. After that preliminary, the execution of CALL statements and the interpretation of function references are described. This provides a basis for the consideration of the special properties of procedures, including recursion and side effects. Next the use of variables and function references to supply the entry point of the invoked procedure is explained. Finally, the details of the syntax of a procedure are given.

ARGUMENTS AND PARAMETERS

Whether a procedure is invoked by a CALL statement or a function reference, an important part of the operation is the passing of arguments to parameters. As a simple example, consider the following program:

```
P:  PROC;
        DCL A FLOAT;
        DCL B FIXED;
        ...
        CALL Q1(A,B,A+6);
        ...
Q1:     PROC(X,Y,Z);
            DCL (X,Y,Z) FLOAT;
            ...
        END;
    END;
```

In this example, the CALL statement invokes the procedure labeled Q1. The CALL statement has a list of three arguments:

(A,B,A+6)

and the procedure has a corresponding list of parameters, one for each argument:

(X,Y,Z)

When the procedure Q1 is invoked, each argument is passed to its corresponding parameter.

The arguments are passed in two ways, as follows:

- If the argument is a variable reference and has a suitable storage type, then the variable designated by the argument is passed; that is, the corresponding parameter is set to designate the same variable as the argument just before procedure execution. Such an argument is a by-reference argument. The first argument in the program just given is a by-reference variable.

- If the argument is suitable for assignment to the corresponding parameter but cannot be handled as a by-reference argument, then the value designated by the argument is passed; that is, a system temporary is allocated, the argument value is assigned to the temporary, and the parameter is set to designate the temporary. Such an argument is a by-value argument. The second argument in the program is by-value because its storage type differs from that of its parameter. The third argument is by-value because it is not a variable reference.

Observe that an argument that is passed by-reference can have its value changed during procedure execution, whereas a by-value argument merely supplies the initial value for a system temporary.

The discussion of arguments and parameters given thus far is intended to be an introduction to the subject. In the following paragraphs, detailed rules and examples are given. First the classification of arguments as by-reference or by-value is defined; then the interpretation of the two kinds of arguments is given. The discussion continues with guidelines for argument usage and concludes with a note on argument validity.

Argument Classification

The rules for the classification of an argument as by-reference or by-value are given here. The classification is performed during program compilation. This treatment contributes to the efficiency of the object program, but it also makes the rules more complicated; specifically, it accounts for the presence of Rules 3 and 4.

An argument is by-reference if it satisfies all of the following rules:

1. A by-reference argument must be a variable reference. This restriction is present because a procedure can assign a value to a by-reference argument; and only a variable reference can be the target of an assignment statement.

2. A by-reference argument must have the same storage type as the corresponding parameter. This restriction is present because a by-reference argument and a parameter designate the same variable and (with exceptions that are not relevant here) every reference to a given variable must use the same storage type.

3. With one exception, a by-reference argument must be declared with constant extents. This restriction is present because argument classification is done at compile time, when the value of a non-constant extent is not known.

   The exception to this restriction allows the use of a non-constant extent in the declaration of an argument if the corresponding extent in the declaration of the parameter is '*'. The '*' parameter extent means that the extent is copied from the argument extent; therefore it necessarily has the same value and there is no need to examine the value of the argument extent.

4. A by-reference argument must not be a reference to an array that is declared as DEFINED and uses isub's in its definition. This restriction is present because a reference to an isub-defined array requires a special encodement, the need for which cannot be detected at compile time.

An argument is by-value if it does not satisfy some of the rules just given and does satisfy the following rule:

5. A by-value argument must have a storage type such that the argument value can be converted, where necessary, to the storage type of the corresponding parameter. This restriction is present because the value of a by-value argument is assigned to a temporary that has the storage type of the parameter.

With one exception, an argument is valid if it can be classified as by-reference or by-value. The exception is:

6. A by-reference argument must not designate an unconnected aggregate if the array bounds of the declaration of the corresponding parameter are all constants. This restriction is present because of efficiency considerations and the compile-time classification of arguments.

The term unconnected is defined in terms of the layout of storage described under "Arrays" in the section on "Value Storage". An array is connected or unconnected, depending on whether its elements are adjacent in storage or not.

EXAMPLES OF ARGUMENT CLASSIFICATION

As the basis for detailed examples of argument classification, consider the following program:

```
P:  PROC;
        DCL 01 S(10) STATIC EXTERNAL,
               02 ACCT DECIMAL(8,2),
               02 ERR ENTRY(FLOAT,FIXED DEC(10)) RETURNS(FLOAT);
        DCL B CHAR(N) CONTROLLED;
        DCL N FIXED;
        DCL 01 T,
               02 ACCT DEC FIXED(8,3),
               02 ERR ENTRY(FLOAT,FIXED DEC(10)) RETURNS(FLOAT);
        DCL C CHAR(30);
        ...
        N = 30;
        ALLOCATE B;
        CALL Q2(S(3),B);
        ...
        CALL Q2(T,C);
        ...
Q2:     PROC(M1,M2);
            DCL 01 M1,
                   02 BALANCE DEC FIXED(8,2),
                   02 EROUT ENTRY;
            DCL M2 CHAR(30);
            ...
            END;
        END;
```

This example has two CALL statements. The arguments in the first CALL statement are classified as follows:

- The argument S(3) is <u>by-reference</u>. The declaration of S is considerably different from that of M1, but a careful examination shows that the declarations give the same storage type: S is an array of structures, but S(3) is a structure of the same aggregate type as M1; the attributes STATIC EXTERNAL are not part of the storage type; DECIMAL(8,2) and FIXED DEC(8,2) are two ways to describe the same data type; and the only part of the declaration of S.ERR that is part of the storage type is the keyword ENTRY.

- The argument B is <u>by-value</u>. Although the storage types of B and M2 are the same when the CALL statement is executed, Classification Rule 3 is not satisfied.

The arguments in the second CALL statement are classified as follows:

- The argument T is <u>by-value</u>. The storage type of T differs from that of M1 in just one place; the scale-factor of T.ACCT is different from that of M1.BALANCE.

- The argument C is <u>by-reference</u>. Its storage type is explicitly the same as that of M2.

This example illustrates the fact that a parameter can correspond to a by-reference argument in one CALL statement and to a by-value argument in another.

EXAMPLES OF CONNECTED AND UNCONNECTED ARGUMENTS

As the basis for examples of arguments that are connected or unconnected arrays, consider the following program:

```
P:  PROC;
        DCL A(12,12) FLOAT;
        ...
        CALL Q5(A(3,*));
        ...
Q5:     PROC(X);
            DCL X(12) FLOAT;
            ...
            END;
        END;
```

The argument A(3,*) is valid for the following reasons:

- It is classified as by-reference.
- It is a cross-section of an array but is nevertheless connected because A(3,1), A(3,2), and so on, are adjacent in storage. Therefore it is a valid by-reference argument for a parameter that has a constant array bound.

Suppose the CALL statement in the example just given is changed to:

```
CALL Q5(A(*,8));
```

This CALL statement is not a valid invocation of the procedure. The argument is not connected because it represents A(1,8), A(2,8), and so on, and the designated variables are not adjacent in storage. There are two ways to correct the call:

- The argument can be parenthesized, thus:

  ```
  CALL Q5((A(*,8)));
  ```

  For purposes of classification, the argument is now a parenthesized expression, not a variable reference. Therefore the argument is by-value and this restriction does not apply. The method cannot be used if the procedure Q5 is programmed to assign a value to A(*,8); and it incurs the added expense of copying the array into a system temporary as part of the passing of the argument.

- The array bound in the parameter can be changed to '*', thus:

  ```
  DCL X(*) FLOAT;
  ```

  The parameter no longer has a constant array bound; therefore, this restriction does not apply. The argument remains by-reference, but it is handled by the more general and more expensive method associated with an '*' array bound.

The choice between these two methods does not arise often because variables that designate unconnected storage are unusual in typical programming applications.

Argument Interpretation

When a CALL statement or function reference is executed, each argument is interpreted according to its classification, as follows:

- If the argument is by-reference, then the corresponding parameter is set, just before each procedure execution, to designate the variable that is currently designated by the argument. The parameter designates that variable throughout the execution of the procedure. Just after procedure execution the association between the parameter and the variable is broken; but the variable itself continues to exist.

- If the argument is by-value, then the interpreter allocates a system temporary that has the same storage type as the parameter, evaluates the argument, assigns the argument value to the temporary, and sets the parameter to designate the temporary. These actions are all taken just before procedure execution. The parameter designates the temporary throughout the execution of the procedure. Just after procedure execution, the interpreter frees the temporary and its value is lost.

The differences in the two interpretations lie entirely in the CALL statement or function reference that invokes the procedure, and not in the interpretation of the procedure itself.

EXAMPLES OF ARGUMENT INTERPRETATION

As the basis for examples of the interpretation of arguments, consider the following program:

```
P:  PROC;
        DCL ALPHA(10) FLOAT;
        DCL BETA FLOAT BIN(40);
        DCL (I,K) FIXED;
        ...
        CALL Q3(ALPHA(I+2),BETA,2*K+6);
        ...
Q3:     PROC(D,X1,X2);
            DCL (D,X1,X2) FLOAT;
            ...
            END;
        END;
```

The arguments in this example are interpreted as follows:

- The argument ALPHA(I+2) is classified as by-reference. Just before procedure execution begins, the subscript expression is evaluated; suppose the current value of the expression is 3. Then the interpreter locates the variable designated by ALPHA(3) and sets the parameter D to designate that variable. During procedure execution, the third element of ALPHA can be referenced either as ALPHA(3) or D. It may happen that the value of I is changed during procedure execution; but that change does not cause D to designate some other element of the array ALPHA. Just after procedure execution, the association between D and ALPHA(3) is broken and D becomes undefined until the next time the procedure is invoked.

- The argument BETA is classified as by-value because its precision is different from that of the parameter X1. Just before procedure execution, the interpreter allocates a system temporary of storage type FLOAT, evaluates the variable reference BETA, and assigns the value to the system temporary. During procedure execution, the parameter X1 designates the temporary, not the variable that is designated by BETA. If the value of BETA is changed during procedure execution, it does not affect the value of the temporary. Just after procedure execution, the temporary is freed.

- The argument 2*K+6 is classified as by-value. It is treated in much the same way as BETA; however, for 2*K+6 it is more readily apparent that a change in the temporary designated by X2 does not change the argument, since a value cannot be assigned to an operator expression.

The Effect of an '*' Extent

When an asterisk, '*', is written as an extent in the declaration of a parameter, it indicates that the extent for the parameter is to be copied from the storage type of the corresponding argument. (An extent is an array bound, a maximum string length, or an area size.) The value of the parameter extent is determined in this way each time the procedure is invoked, and therefore can change from one invocation to another. Under certain circumstances, when the aggregate type of the argument must be converted, there is no argument extent that corresponds to a '*' parameter array bound; then, the value 1 is assumed for the array bound.

AN EXAMPLE OF AN '*' EXTENT

As an example of the use of an '*' as an array bound, consider the following program:

```
P:  PROC;
        DCL A(S) FLOAT CONTROLLED;
        DCL S FIXED;
        ...
        ALLOCATE A;
        CALL Q4(A);
        ...
        CALL Q4(200);
        ...
Q4:     PROC(X);
            DCL X(*) FLOAT;
            ...
        END;
    END;
```

This program has two CALL statements, and their arguments are passed as follows:

- In the first call, the argument is by-reference. The details of the program are not shown, but suppose this CALL statement is executed twice, with S equal to 10 and 12, respectively. The first time, A is allocated with ten elements and the declaration of the parameter becomes X(10) FLOAT. The second time A has twelve elements and the declaration is X(12) FLOAT.

- In the second call, the argument is by-value. The argument must be converted from a FIXED DEC(3) scalar to a FLOAT array. Since the array bound is given neither by the argument nor the parameter, it is assumed to be one. Therefore, the declaration of the parameter becomes X(1) FLOAT.

Guidelines for Arguments

The classification of arguments is performed automatically by the compiler, without an explicit indication in the program. This convenient arrangement must be supervised carefully by the programmer, and he should be aware of the classification of each argument as he writes a CALL statement. The effect of classification on both efficiency and correctness of a program is considered here.

First consider efficiency. The cost of passing an argument becomes significant when the storage type of the argument has one or more extents; such an argument designates a relatively complicated and potentially large variable. If the argument is by-reference, it is necessary only to set the parameter to designate the argument variable; this is an inexpensive operation, whose cost is independent of the extents of the variable. On the other hand, if the argument is by-value, a temporary must be allocated and the entire value copied into the temporary; and this can be expensive. A purpose of the '*' parameter extent is to allow the passing of arguments by-reference that would otherwise be passed by-value because their extents did not agree with those chosen for the parameter.

Next consider correctness. A serious mistake is the assumption that an argument is by-reference when it is not; the assumption is easily made when the argument is a variable reference. Suppose the procedure Q4 in the most recently given example delivers a value by assigning it to the parameter, X. This is correct because the argument A is by-reference and will receive the delivered value. Suppose, however, that the programmer decides, at the last minute, to increase the precision of A and therefore changes its declaration to:

```
DCL A(S) FLOAT(40) CONTROLLED;
```

Now the storage type of the argument is different from the parameter and the argument is classified as by-value. Instead of changing the value of A, the procedure changes the value of a system temporary, and the delivered value is lost. This mistake is serious because it is not detected by the compiler and is not apparent to a casual reader.

Whenever the compiler can detect a difference between storage types of argument and parameter, it issues a warning to let the programmer know that a by-value passing occurs. The compiler detects such a mismatch when the parameter is for an internal procedure or for an external procedure for which the programmer has supplied an entry declaration specifying the type of each parameter for the procedure. Since the external procedure is compiled separately, however, the compiler is <u>not</u> able to check to see that the entry declaration does correctly specify the type of each parameter. See the discussion of "Entry References and Declarations" later in this section.

THE CALL STATEMENT

A CALL statement has one of the following forms:

```
CALL  ref(  arglist  );

CALL  ref();

CALL  ref;
```

where ref is the entry reference and arglist is the argument list. The entry reference must yield a scalar entry value. The argument list is a sequence of arguments separated by commas, and each argument is an expression. As special cases, the argument list can be empty, as shown in the second form, or the parenthesized argument list can be absent, as shown in the last form; these special cases are equivalent and are used when no arguments are required.

Agreement of CALL with Entry Point

The entry reference of a CALL statement designates a procedure entry point. The form of the CALL statement must agree with the form of the procedure entry point. Specifically, the CALL statement must satisfy the following restrictions:

1. The number of arguments in the CALL statement must be equal to the number of parameters at the designated procedure entry point. This restriction provides for the matching of arguments with parameters. If there are no arguments in the CALL statement, then there must be no parameters at the designated procedure entry point.

2. Each argument must be a valid by-reference argument or a valid by-value argument as defined earlier in the discussion of classification under "Arguments and Parameters".

3. A RETURNS attribute must not appear at the designated procedure point. This restriction is present because a RETURNS attribute only makes sense if the procedure is invoked by a function reference.

The Execution of CALL Statements

The execution of a CALL statement is performed in five steps, as follows:

```
Interpret entry reference
Interpret arguments
Activate procedure
Execute procedure
Exit from procedure
```

These steps are now considered in detail.

INTERPRET ENTRY REFERENCE

The first step in the execution of a CALL statement is the interpretation of the entry reference, as follows:

1. The entry reference is evaluated to yield an entry value.

2. The entry value is used to locate a procedure entry point. In a PL/I procedure, the procedure entry point is either a PROCEDURE statement or an ENTRY statement. The procedure that immediately contains the designated procedure entry point is the invoked procedure for this execution of the CALL statement.

The entry referenced may be in the same external procedure as the reference (as when the entry is to an internal procedure or to the external procedure itself). Or the entry referenced may be in an external procedure different from the one including the reference. Such a different external procedure may or may not be written in PL/I and may or may not have been written by the programmer writing the procedure with the reference.

INTERPRET ARGUMENTS

The next step in the execution of a CALL statement is the interpretation of arguments, which was described in detail under "Arguments and Parameters". Briefly,

- Each by-reference argument is interpreted by evaluating any subscript expressions or locator-qualifier expressions that appear in the argument. Then the designated variable is located and the parameter that corresponds to the argument is set to designate that variable.

- Each by-value argument is evaluated. Then a system temporary is allocated that has the same storage type as the corresponding parameter, the value of the argument is assigned to the temporary, and the parameter is set to designate the temporary.

The order in which the arguments are interpreted is not defined. After this step, each parameter designates either a variable or a temporary.

ACTIVATE PROCEDURE

The third step is the activation of the procedure. A new activation region is created for the procedure (in the language of GCOS PL/I, a new stack frame is added to the stack). Storage units are allocated in this activation region as follows:

- Each variable of storage class AUTOMATIC that is declared in the invoked procedure is allocated. Since the AUTOMATIC is assumed when no storage class attribute is given, most variables of a typical program are handled in this way.

- A system temporary for the return address is allocated. The return address is used to resume execution after the CALL statement when execution of the invoked procedure is complete.

- Other system temporaries necessary for maintaining the stack, handling ON conditions, holding temporary results, and so on are allocated. These need not be considered in detail since the programmer does not refer to them directly.

In order to reduce execution cost, the GCOS implementation of PL/I avoids the creation of a block activation region for certain procedures. This optimization technique does not change the interpretation of a program, and it can be ignored by a programmer who remains within the framework of the PL/I language. However, when the programmer uses various routines to examine stack frames directly, the effects of the technique are apparent. The compiler prints a message for each procedure that is compiled without an activation region, and provides information about the distribution of the storage units that would have occupied the missing activation region.

EXECUTE PROCEDURE

The fourth step is the execution of the procedure. Execution begins with the PROCEDURE or ENTRY statement that is at the procedure entry point. The execution of this statement causes no action because its only purpose is to describe a procedure entry point; but it provides the starting point for sequential execution of statements of the procedure.

Execution of the procedure continues until an exit statement is executed, as described in the next step.

EXIT FROM PROCEDURE

The last step in the execution of a CALL statement is the execution of an exit statement. Each exit statement specifies an exit destination; that is, the statement that is excuted once the execution of the CALL statement is complete. There are three kinds of exit statement, as follows:

- A RETURN statement is an exit statement for the given procedure if it is contained in that procedure but not in a smaller procedure. Its exit destination is the statement that appears next after the given CALL statement.

- An END statement is an exit statement for the given procedure if it is the last statement of that procedure. Its exit destination is also the statement that appears next after the given CALL statement.

- A GOTO statement is an exit statement for the given procedure if it is part of the given procedure or of a more recently invoked procedure and its destination lies outside the given procedure. The exit destination is the destination of the GOTO statement.

If the exit statement is a RETURN statement, it must not have a returned result; that is, it must have the form:

```
RETURN;
```

This restriction reflects the fact that a CALL statement does not expect a returned value.

As part of the execution of the exit statement, the following operations are performed:

- The invoked procedure is deactivated; that is, the variables and temporaries that are allocated in the activation region for the procedure are freed and the activation region is discarded. (In the language of GCOS PL/I, the stack frame associated with the procedure is removed from the stack.)

- Each system temporary allocated for a by-value argument is freed.

After these operations, control is transferred to the statement designated by the exit destination.

Examples of CALL Statements

As an example of the execution of a CALL statement, consider the following program:

```
P:  PROC OPTIONS(MAIN);
        DCL ALPHA FLOAT;
        ...
        CALL RA(ALPHA+1);
        ...
LAB:    ...
        ...
RA:     PROC(X);
            DCL X FLOAT;
            ...
            CALL RB(X);
            ...
            END;
RB:     PROC(Y);
            DCL Y FLOAT;
            DCL A(20) FLOAT;
            ...
            GOTO LAB;
            ...
            END;
    END;
```

The external procedure P is passed control by GCOS. This execution is equivalent to the following PL/I statement:

```
CALL P;
```

Execution of the command proceeds as follows:

1. Interpret Entry Reference to P. The P in the CALL statement designates the first statement of the program.

2. Interpret Arguments for P. There are no arguments.

3. Activate P. An activation region for the procedure named P is created. The AUTOMATIC variable ALPHA and the required temporaries are allocated in that region.

4. Execute P. Execution begins at the designated PROCEDURE statement and proceeds until the first CALL statement is reached. The execution of the CALL statement proceeds as follows:

    1. Interpret Entry Reference to RA. The RA in the CALL statement designates the second PROCEDURE statement in the program.

    2. Interpret Arguments for RA. The argument ALPHA+1 is by-value. A system temporary of storage type FLOAT is allocated, the value of the argument is assigned to it, and the parameter X is set to designate the temporary.

    3. Activate RA. An activation region for the procedure named RA is created and the required system temporaries are allocated in it.

    4. Execute RA. Execution begins with the PROCEDURE statement labeled RA and proceeds until the second CALL statement of the program is reached. The execution of the CALL statement proceeds as follows:

        1. Interpret Entry Reference to RB. The RB in the CALL statement designates the third procedure statement in the program.

        2. Interpret Arguments for RB. The argument X is by-reference. The parameter Y is set to designate the storage unit designated by X, which is the system temporary created in Step 2, above.

        3. Activate RB. An activation region for the procedure named RB is created, and the array A and the system temporaries are allocated in it.

        4. Execute RB. Execution begins with the PROCEDURE statement labeled RB and proceeds until the GOTO statement is reached. This statement is the exit statement for RB because its destination is outside of RB. It is also the exit statement for RA because, even though it is not contained in RA it is contained in a more recently invoked procedure.

        5. Exit from RB. The procedure RB is deactivated by freeing A and its temporaries and discarding its activation region. Its argument is by-reference, so there is no argument temporary to free. This concludes execution of the statement CALL RB(X);.

    5. Exit from RA. The procedure RA is deactivated by freeing its temporaries and discarding its activation region. The storage temporary used for its argument is freed. This concludes execution of the statement CALL RA(ALPHA+1);.

5. Continue Execution of P. Execution of the external procedure continues, starting with the statement labeled LAB and proceeds until the END statement that is the last statement of the program is reached. This statement is the exit statement for P.

6. Exit from P. The procedure P is deactivated by freeing ALPHA and its temporaries and discarding the activation region. This concludes execution of the program.

The most important feature of this example is the fact that the GOTO statement serves as the exit statement for both the procedure named RB and the procedure named RA.

FUNCTION REFERENCES

A function reference has one of the following forms:

ref(arglist)

ref()

where ref is the entry reference and arglist is the argument list. The entry reference must yield a scalar entry value. The argument list is a sequence of arguments separated by commas, and each argument is an expression. The second form is used when no arguments are required.

Agreement of Reference with Entry Point

The entry reference of a function reference designates a procedure entry point. The form of the function reference must agree with the form of the procedure entry point as follows:

1. The number of arguments in the function reference must be equal to the number of parameters at the designated procedure entry point.

2. Each argument must be a valid by-reference argument or a valid by-value argument as defined earlier in the discussion of classification and validity under "Arguments and Parameters".

3. A RETURNS attribute must appear at the designated procedure entry point. The attribute is necessary because it specifies the storage type of the result returned by the function and thus specifies the storage type of the function reference itself.

When a procedure is invoked by the interpretation of a function reference, it is sometimes called a function; however, that terminology is not used here.

The Interpretation of Function References

The interpretation of a function reference has six steps, as follows:

Interpret entry reference
Interpret arguments
Activate procedure
Execute procedure
Exit from procedure
Fetch result

The first four steps are the same as for the execution of a CALL statement. The fifth step is different and the sixth step is new. These last two steps are now considered in detail.

EXIT FROM PROCEDURE

The fifth step in the interpretation of a function reference is the execution of an exit statement. There are only two kinds of exit statement for a function reference, as follows:

- A RETURN statement is an exit statement for the given procedure if it is contained in that procedure but not in a smaller procedure. Its exit destination is that point in the program at which the value of the function reference is about to be used.

- A GOTO statement is an exit statement for the given procedure if it is part of the given procedure or of a more recently invoked procedure and its destination lies outside the given procedure. The exit destination is the destination of the GOTO statement. In this case, the procedure does not return a value.

If the exit statement is a RETURN statement, it must have a returned result; that is, it must have the form:

RETURN(*r*);

where *r* is an expression. The END statement that is the last statement of the given procedure is not a valid exit statement and must not be executed. Both of these restrictions reflect the fact that a procedure that is invoked by a function reference must return a value unless it exits with a GOTO statement.

As part of the execution of the exit statement, the following operations are performed:

- The invoked procedure is deactivated; that is, the variables and system temporaries that were allocated in the activation region for the procedure are freed and the activation region is discarded.

- Each system temporary that was allocated for a by-value argument is freed.

- The result is returned. That is, a system temporary is allocated with the storage type given by the RETURNS attribute, the expression in the RETURN statement is evaluated, and the value is assigned to the system temporary. More is said of the way a result is returned later in this section under "The RETURNS Attribute".

After these operations, control is transferred to the point designated by the exit destination.

FETCH RESULT

As the last step in the interpretation of a function reference, the result is fetched from the temporary to which it was assigned by the preceding step. This value is the value of the function reference.

An Example of a Function Reference

As an example of the interpretation of a function reference, consider the following program:

```
P:  PROC;
        DCL (ALPHA,BETA) FLOAT;
        ...
        ALPHA = 2*F1(BETA);
        ...
F1:     PROC(X) RETURNS(FLOAT);
            DCL X FLOAT;
            DCL Y FIXED;
            ...
            RETURN(Y+1);
            ...
        END;
    END;
```

The execution of the assignment statement begins with the interpretation of the function reference, as follows:

1. Interpret Entry Reference. The F1 in the function reference designates the second PROCEDURE statement in the program.

2. Interpret Arguments. The argument BETA is by-reference. The parameter X is set to designate the storage unit designated by BETA.

3. Activate Procedure. An activation region for the procedure named F1 is created. The AUTOMATIC variable Y and all required temporaries are allocated.

4. Execute Procedure. Execution begins with the PROCEDURE statement labeled F1 and proceeds until the RETURN statement is reached.

5. Exit from Procedure. The expression Y+1 is evaluated; then the resulting value is converted to FLOAT and assigned to the system temporary that is provided for it. The procedure F1 is deactivated by freeing and discarding its activation region.

6. Fetch Result. The result is fetched from storage and becomes the value of the function reference F1(BETA).

After the function reference is interpreted its result is multiplied by two and assigned to ALPHA.

PROCEDURES

A procedure gathers a sequence of statements together for execution as if they were a single operation. A procedure is defined as:

- A PROCEDURE statement, followed by
- A sequence of statements that is the body of the procedure and that may include some ENTRY statements and RETURN statements, followed by
- An END statement.

Procedures, BEGIN blocks, and DO groups must be nested with respect to one another, as described earlier, in the section on "Program Syntax".

Many examples of procedures are given throughout this section, and therefore no additional examples are given here. Instead, the specialized statements used in procedures are described, beginning with the PROCEDURE statement, continuing with the ENTRY statement, and concluding with the RETURN and END statements.

The PROCEDURE Statement

A PROCEDURE statement has one of the following forms:

px PROCEDURE(parlist) [RETURNS(rd)] [RECURSIVE] [OPTIONS(opt)] ;

px PROCEDURE( ) [RETURNS(rd)] [RECURSIVE] [OPTIONS(opt)] ;

px PROCEDURE [RETURNS(rd)] [RECURSIVE] [OPTIONS(opt)] ;

where px is the prefix, PROCEDURE can be abbreviated as PROC, parlist is the parameter list, rd is the result descriptor and opt is the option. The brackets indicate that the RETURNS attribute, the RECURSIVE keyword, and the OPTIONS attribute are optional. The second and third forms are equivalent and are used for an entry point with no parameters.

The execution of a PROCEDURE statement causes no action; its purpose is to indicate the beginning of a procedure and to define an entry point to a procedure. Descriptions of the prefix, the parameter list, the RETURNS attribute, the RECURSIVE keyword, and the OPTIONS attribute follow.

The prefix of a PROCEDURE statement is a sequence of any number of condition prefixes followed by a sequence of one or more label prefixes. Each condition prefix applies to the entire procedure that begins with the PROCEDURE statement; the effects of these condition prefixes are described later, in the section on "Condition Handling". Each label prefix is an identifier followed by a colon, and its effect is described here.

An identifier in a label prefix in a PROCEDURE statement is a procedure name, and it is interpreted as follows:

- A procedure name has the attributes INTERNAL ENTRY CONSTANT or EXTERNAL ENTRY CONSTANT, depending on whether it is at the beginning of a procedure that is contained in a larger procedure or not.
- A PROCEDURE statement begins a given procedure, but each label prefix that begins the PROCEDURE statement is considered to be outside the procedure. Thus, the declaration of an INTERNAL procedure name is established in the procedure or BEGIN block that immediately contains the given procedure; and the declaration of an EXTERNAL procedure name is established in an imaginary BEGIN block that encloses all of the external procedures of a program.
- During execution of the program, a reference to the procedure name yields the ENTRY value that designates the PROCEDURE statement.

The prefix of a PROCEDURE statement usually consists of just a single label prefix, as in the following example:

```
P3:  PROC(X);
```

During the informal discussion of programs, it is convenient to use the identifier in a label prefix both as a name for a procedure and as a name for an entry point of the procedure. For example, it might be said of the statement CALL P3(A+1); that it "invokes the procedure P3" or that it "invokes a procedure at entry point P3". Strictly speaking, however, the statement "invokes a procedure at the entry point that is designated by the ENTRY value that is associated with the name P3".

The following PROCEDURE statement has a prefix that consists of three label prefixes:

```
ALPHA:
TOP_RUN:
F13:    PROC(Y);
```

The three ENTRY constant names, ALPHA, TOP_RUN, and F13, all designate the same entry point. The layout used in this example is recommended because it places each name at the left margin and makes it easy for a reader to search for an entry name.

THE PARAMETER LIST

The parameter list is a sequence of <u>parameters</u> separated by commas, and each parameter is an identifier. Parameters should be declared; and, for clarity, the declarations of the parameters should be the first statements of the procedure body.

```
AA:  PROC(ALPHA,MAX_VAL);
        DCL ALPHA FIXED DEC(8);
        DCL MAX_VAL  FLOAT;
        ...
        END;
```

With the following exceptions, a parameter name is declared like any other variable name:

- When an identifier appears in a parameter list, its scope, storage class, and category must be INTERNAL, PARAMETER, and VARIABLE, respectively. Since these attributes are supplied by default, there is no need to write them in the declaration of a parameter.

- The INITIAL attribute cannot be used. This exception is present because the INITIAL attribute is valid only for a variable that has its own storage, and a parameter is always set to designate previously allocated storage.

- An extent (array bound, maximum string length, or area size) can be written only as an '*' or as a decimal integer constant. The use of the '*' extent is discussed earlier, under "Arguments and Parameters"; it indicates that the extent for the parameter is to be copied from the storage type of the corresponding argument. A decimal integer constant extent does not provide for anything that an '*' cannot do, but it permits more efficient compilation of references to the parameter, and should be used where the generality of an '*' extent is not required.

These exceptions reflect the logical consequences of the way in which parameters are interpreted; they do not restrict the parameters. Indeed, an important feature of PL/I is that any value, scalar or aggregate, computational or non-computational, can be passed as an argument to a procedure or returned as a result.

THE RETURNS ATTRIBUTE

Depending on whether the RETURNS attribute is present or absent, an entry point is invoked by a function reference or a CALL statement, respectively. The RETURNS attribute has the following form:

```
RETURNS( d )
```

where <u>d</u> is the descriptor. The descriptor gives the declaration of the storage type of the returned result; with the following exceptions, it is written exactly as the storage type of a variable is given in a DECLARE statement:

- Any attributes that are not part of the storage type are omitted.

- The names of members of a structure are omitted.

- Each extent (array bound, maximum string length, or area size) must be either an '*' or a decimal integer constant.

The storage type given by the descriptor of the RETURNS attribute describes the system temporary to which the RETURN statement assigns its value. This temporary is associated with the function reference that invoked the procedure.

An Example of the RETURNS Attribute

For a rather complicated example of a descriptor, consider the following procedure:

```
A1:  PROC(X) RETURNS(01 DIM(*),
                        02 FLOAT,
                        02 FIXED DEC(8));
          DCL 01 X(*),
                 02 NUM FLOAT,
                 02 M FIXED DEC(8);
          RETURN(X+1);
          END;
```

This procedure accepts an array of structures with any bounds as its parameter and it returns a result that has exactly the same storage type. It is chosen as an example because the declaration of X, given in a DECLARE statement, can be compared to the declaration of the result, given as a descriptor in the RETURNS attribute.

THE RECURSIVE KEYWORD

The RECURSIVE keyword has the form:

RECURSIVE

This keyword applies to an entire procedure, not to a particular entry point; therefore, it is used only in a PROCEDURE statement and is not called an attribute. Furthermore, the keyword refers to the way in which a procedure is invoked and not to the action taken by the procedure.

A procedure is <u>recursive</u> if, for some possible execution of the program in which it appears, it is invoked when a previous invocation is still active; examples are given later, under "Recursive Procedure Execution". In Standard PL/I, every recursive procedure must have the keyword RECURSIVE in its PROCEDURE statement. In some implementations of PL/I, the addition of the keyword causes a change in the way the procedure is compiled; specifically the code becomes valid for recursion but less efficient. The GCOS implementation of PL/I assumes that every procedure is recursive and thus does not depend on the RECURSIVE keyword. The appropriate use of the RECURSIVE keyword is recommended to GCOS programmers because it maintains compatibility with the Standard and gives useful information about the procedure to a reader.

THE OPTIONS ATTRIBUTE

The OPTIONS attribute on a PROCEDURE declaration has only one form:

OPTIONS (MAIN)

(For other OPTIONS attributes accepted on ENTRY declarations, see the discussion of "Entry References and Declarations" later in this chapter.) The OPTIONS (MAIN) attribute can only be given on an external procedure declaration. It identifies the procedure to which GCOS is to pass control when the program is loaded and execution is initiated. If none of the external procedures that call one another, and hence are loaded together, are given the OPTIONS (MAIN), the first loaded procedure is the one to which control is passed when execution is initiated. If more than one procedure is given the OPTIONS (MAIN) attribute, those after the first are remarked as erroneous and control is transferred to the first.

The name of an external procedure may also be given on a $ ENTRY control card that is processed by the loader to identify which of several procedures being loaded is to be given control when execution is initiated.

The procedure to which control is given should not specify any parameters, for no arguments are passed when control is given.

The ENTRY Statement

An ENTRY statement has one of the forms:

px ENTRY(parlist) [RETURNS(rd)] ;

px ENTRY( ) [RETURNS(rd)] ;

px ENTRY [RETURNS(rd)] ;

where px is the prefix, parlist is the parameter list, and rd is the result descriptor. An ENTRY statement is the same as a PROCEDURE statement except for the following points:

- The ENTRY statement does not appear at the beginning of a procedure; instead, it can occur at any point in the body of the procedure. It is used when more than one entry point is required for a procedure.
- The prefix of an ENTRY statement must not contain a condition prefix.

The last exception is present because the condition prefixes are given, once and for all, in the PROCEDURE statement that begins the procedure.

The execution of an ENTRY statement causes no action; its sole purpose is to define an entry point to a procedure. Thus when an ENTRY statement is encountered by sequential flow of control through the procedure, the ENTRY statement has no effect and control continues to the next statement.

AN EXAMPLE OF A MULTIPLE-ENTRY PROCEDURE

Two or more procedures can be usefully combined into a single procedure when they have statements or data in common. The following procedure has two entries, one for use with two arguments, and the other for use with one argument.

```
P2A: PROC(X,YPAR);
          DCL (X,YPAR) FLOAT;
          DCL Y FLOAT;
          Y = YPAR;
          GOTO START;
P1A:      ENTRY(X);
          Y = 1;
START:    ...
          END;
```

This procedure has two entries. The first entry has two arguments. The second entry has only one argument and provides the value 1 for the second argument. The handling of the parameters seems awkward, but the following simpler version is not valid.

```
P2A: PROC(X,Y);
          DCL (X,Y) FLOAT;
          GOTO START;
P1A:      ENTRY(X);
          Y = 1;
START:    ...
          END;
```

This program is invalid because when it is invoked at its second entry, P1A, it uses the parameter Y which is not defined for that entry.

The RETURN Statement

The RETURN statement has one of the following forms:

```
RETURN( re );

RETURN;
```

where re is the return expression. The first form is a valid exit statement for a procedure that was invoked by a function reference; the second form, for a procedure that was invoked by a CALL statement.

The END Statement

The END statement has the following form:

```
END;
```

The purposes of the END statement are to indicate the end of a procedure (or a BEGIN block or a group) and to serve as an exit statement for a procedure. An END statement is a valid exit statement for a procedure only if the procedure was invoked by a CALL statement.

ENTRY REFERENCES AND DECLARATIONS

The entry reference of a CALL statement or a function reference specifies the entry point for the invoked procedure. In order to process the entry reference completely, however, the compiler must be supplied additional information, including the storage types of any parameters and return value. This information is supplied, basically, in one of two ways: in an ENTRY or PROCEDURE statement; or, in a DECLARE statement. The manner in which it is supplied depends on the kind of entry reference that is used.

An entry reference is usually a constant reference to an entry point in the same external procedure. That case is handled in a simple way in PL/I, and the examples given thus far have all used that kind of entry reference. In what follows, the constant entry reference is given further consideration and then other kinds of entry references are introduced.

Constant Entry References

An entry reference can designate an entry point in the same external procedure, in a different external procedure, or in a non-PL/I procedure. Each case requires special consideration when the entry reference is a constant reference.

ENTRY POINTS IN THE SAME EXTERNAL PROCEDURE

When a constant entry reference designates an entry point that is within the same procedure, the compiler can obtain the information it requires from the designated PROCEDURE statement or ENTRY statement. Therefore, no additional declaration is required. This consideration applies to external entry names in the same procedure as well as internal entry names.

ENTRY POINTS IN ANOTHER EXTERNAL PROCEDURE

When a constant entry reference designates an entry point that is in another external procedure of the program, the compiler cannot obtain information directly about the parameters and the returned value because external procedures are compiled separately. Therefore, the programmer should consult the invoked entry point, obtain the information, and declare the entry name in the procedure in which it is called by means of a DECLARE statement.

Use of the DECLARE statement in this case is required for a function reference. It is not required for a CALL reference, but it is strongly advised on the basis of earlier discussions under "Guidelines for Arguments", in this section, and "Constant References" in the section on "Expressions".

The DECLARE statement for the ENTRY constant name must have the following attributes:

- An ENTRY attribute with parameter descriptors. The keyword ENTRY is followed by a parenthesized list of descriptors, one for each of the parameters at the entry point. The descriptor is obtained from the declaration of the parameter by deleting attributes that are not part of the storage type and deleting any names of members of a structure.

- The RETURNS attribute (if one appears at the invoked entry point).

When an ENTRY name is declared in this way, its scope and category are assumed to be EXTERNAL CONSTANT.

As an example of the declaration of a constant name that designates an entry point in another external procedure, consider the following program:

```
P:  PROC;
        DCL B3 ENTRY(01 DIM(*),
                       02 FLOAT,
                       02 FLOAT,
                    FIXED)
               RETURNS(FLOAT);
        DCL (I,J) FIXED;
        ...
        I = B3(0,J);
        ...
        END;

    B3: PROC(A,B) RETURNS(FLOAT) RECURSIVE;
            DCL 01 A(*),
                   02 Q1 FLOAT,
                   02 Q2 FLOAT;
            DCL B FIXED;
            ...
            END;
```

Because of the DECLARE statement at the beginning of the first external procedure, P, the compiler can compile the procedure separately and yet produce the correct code for the statement I=B3(0,J);.

ENTRY POINTS IN A NON-PL/I PROCEDURE

When a constant entry reference designates an entry point of a procedure written in a language other than PL/I, the same constant entry declaration as used for entry points in another PL/I external procedure is needed. However, there are special OPTIONS attributes which may also be required.

The reason for the OPTIONS attribute in an entry declaration is to instruct the compiler to provide for a different <u>way</u> of passing arguments to the entry. (Concerning the question of appropriate storage types to be used for the parameter and result descriptors, refer to the PL/I User's Guide in the sections on "Linking PL/I and Other Languages" or "Internal Representation of PL/I Data".)

Additional OPTIONS Attributes

The OPTIONS attribute has already been discussed in connection with the PROCEDURE statement. The OPTIONS attributes described here, however, can appear only in the declaration of non-PL/I entry points. The OPTIONS attribute, in this case, has one of the following forms:

```
OPTIONS (VARIABLE)

OPTIONS (GMAP)
```

OPTIONS (FORTRAN)

OPTIONS (COBOL)

The OPTIONS (VARIABLE) attribute indicates that the entry may be called at different times with a different number of arguments or arguments of different storage types, or both. In such a case, the compiler generates instructions to pass not only the arguments but also descriptors for each argument that identify the storage type and scale, precision, extent, etc., of each argument (or each member of a structure argument).

If no parameter attributes are provided on an ENTRY declaration and a call is made to that entry with arguments, the compiler issues a warning to note that the entry is treated as if OPTIONS (VARIABLE) were specified for it and provides the descriptors that this attribute indicates.

The OPTIONS (GMAP), OPTIONS (FORTRAN), or OPTIONS (COBOL) attribute indicates that the entry expects arguments to be passed in a way specific to the attributed language and different than expected by a PL/I procedure. In such cases, the PL/I compiler generates instructions to pass arguments in a way appropriate to the language.

Both the OPTIONS (VARIABLE) and a language option, say OPTIONS (GMAP), may be attributed to the same entry by writing both options, for example:

OPTIONS (GMAP) OPTIONS (VARIABLE)

or

OPTIONS (VARIABLE) OPTIONS (GMAP).

In such cases, arguments and descriptors are passed in a way appropriate to the language. Naturally, the procedure written in the other language must expect descriptors to accompany the arguments.

Variable Entry References

When an entry reference is a variable reference, the compiler cannot obtain information directly about parameters and RETURNS attribute because it cannot predict which entry point will be designated by the entry reference. Therefore, the programmer must include the required information in the declaration of the variable name.

The declaration of an entry variable reference must inciude the following attributes:

- An ENTRY attribute whose parameter descriptors agree with the parameter declarations of every entry point that will be designated by the entry reference.
- A RETURNS attribute that is identical to that given (if any) for every entry point that will be designated by the entry reference.
- The category attribute VARIABLE.

The category attribute must be given because the default is CONSTANT when the data type is ENTRY. The scope, storage class, and INITIAL attributes can be chosen according to considerations that would govern the declaration of any kind of variable.

During execution of the program, the evaluation of a variable entry reference must yield an ENTRY value that designates a valid entry point.

AN EXAMPLE OF AN ENTRY VARIABLE REFERENCE

As an example of the use of an entry variable reference, consider the following program:

```
P:  PROC;
        DCL V ENTRY(DIM(10) FLOAT,FIXED)
            RETURNS(FLOAT) STATIC VARIABLE;
        DCL VX ENTRY VARIABLE;
        DCL A(10) FLOAT;
        DCL (M,N) FIXED;
        ...
        IF M=0 THEN VX = F1; ELSE VX = F2;
        ...
        V = VX;
        ...
        N = 3*V(A,3)+1;
        ...
F1:     PROC(A1,B1) RETURNS(FLOAT) RECURSIVE;
            DCL A1(10) FLOAT;
            DCL B1 FIXED;
            ...
            END;
F2:     PROC(A2,B2) RETURNS(FLOAT) RECURSIVE;
            DCL A2(10) FLOAT;
            DCL B2 FIXED;
            ...
            END;
        END;
```

The program uses ENTRY variables as follows:

- At the top of the program, V is declared as an ENTRY STATIC VARIABLE without an initial value. The remainder of the declaration is included not to describe the storage used for the variable but rather to allow the use of the variable as an entry reference.

- Next VX is declared ENTRY. Since this variable is not used as an entry reference, parameter descriptors and RETURNS attribute are omitted from its declaration.

- At some time during the program, the IF statement assigns the ENTRY value designated by the constant name F1 or F2 to VX.

- Later, the value of VX is assigned to V.

- Each time the assignment to N is encountered, the variable name V is used as an entry reference. The declaration of V allows the compiler to determine whether the arguments are by-name or by-value and to determine the storage type of the returned result (and thus of the function reference V(A,3) itself).

Function Entry References

When an entry reference is a function entry reference, the programmer must include parameter descriptors and RETURNS attribute in the RETURNS attribute of the function name.

AN EXAMPLE OF AN ENTRY FUNCTION REFERENCE

As an example of the use of an entry function reference, consider the following program:

```
P:  PROC;
        DCL A(10) FLOAT;
        DCL (M,P) FIXED;
        ...
        P =.3*E(M)(A,3)+1;
        ...
E:      PROC(A) RETURNS(ENTRY(DIM(10) FLOAT,FIXED)
                        RETURNS(FLOAT));
            DCL A FIXED;
            IF A=0 THEN RETURN(F1); ELSE RETURN(F2);
            END;
F1:     PROC(A1,B1) RETURNS(FLOAT) RECURSIVE;
            DCL A1(10) FLOAT;
            DCL B1 FIXED;
            ...
            END;
F2:     PROC(A2,B2) RETURNS(FLOAT) RECURSIVE;
            DCL A2(10) FLOAT;
            DCL B2 FIXED;
            ...
            END;
END;
```

Compare this program to the example given for variable entry references. In this program, the selection between F1 and F2 still depends on M, but it is carried out by a function reference, E(M).

Generic Entry Names

A generic entry name is used to designate one of a set of entry points. The programmer specifies, in the declaration of the generic entry name, how a single entry point is selected for a particular use of the name. The selection is done during compilation, and it is based on the attributes of the arguments that accompany a particular use of the name.

AN EXAMPLE OF A GENERIC ENTRY NAME

As an example of the declaration and use of a generic entry name, consider the program:

```
P:   PROC;
         DCL Q GENERIC (Q1 WHEN (FIXED), Q2 WHEN (FLOAT));
         DCL Q1 ENTRY (FIXED), Q2 ENTRY (FLOAT);
         DCL A FIXED, B FLOAT;
         ...
         CALL Q (A);
         ...
         CALL Q (B);
         ...
Q1:      PROC (N);
             DCL N FIXED;
             ...
             END;
Q2:      PROC (N);
             DCL N FLOAT;
             ...
             END;
         END;
```

Presumably, procedures Q1 and Q2 perform essentially the same service but for different types of arguments, one for fixed and the other for float, respectively. During the compilation of the program, the CALL Q(A) is replaced by CALL Q1(A), and the CALL Q(B) by CALL Q2(B). Thus all references to Q are eliminated.

The full description of the generic entry name is given in the PL/I Language Manual. The facility is not described fully here because its usefulness is limited.

RECURSIVE PROCEDURE EXECUTION

Sometimes a procedure is invoked when a previous invocation is still active. Such a procedure is recursive. Recursion can be direct or chained. Direct recursion results if the body of a given procedure contains a CALL statement or function reference that invokes the given procedure. Chained recursion occurs when a procedure A invokes a procedure B that invokes a procedure C and so on until a procedure invokes procedure A. In either case, the keyword RECURSIVE should be used in the PROCEDURE statement at the beginning of the procedure.

When a procedure is executed recursively, the procedure has more than one activation internal region. These are the storage regions that, in the GCOS implementation of PL/I, are the stack frames. An activation region is used for AUTOMATIC variables, for the extent values of DEFINED variables, for the pointers associated with PARAMETER variables, and for various system temporaries required for the evaluation of expressions, return from a procedure, and so on. A complete description of the role of activation regions is given earlier, in the section on "Storage Management".

The description of recursion that follows covers two kinds of recursion, surface and general. Surface recursion is a slightly restricted form of recursion whose rules are much easier to learn than those for general recursion. General recursion is, by definition, the unlimited use of recursion in PL/I and includes surface recursion as a special case. As part of the description of these two kinds of recursion, many examples are given. Most of the examples are chosen to illustrate the rules of recursion, not to provide examples of the practical use of recursion. However, the description of surface recursion ends with a practical program that is a very simple but otherwise realistic compiler.

The subject of recursion is not of equal interest to all programmers, and the following remarks describe three different levels of interest:

- Most scientific programs and nearly all business programs do not use recursion at all. If the reader does not plan to write recursive programs, he can skip the remainder of this section.
- Surface recursion is adequate for many recursive problems. Surface recursion excludes certain uses of statement address variables and certain uses of ON statements. If the reader does not plan to go beyond surface recursion, then he can skip the description of general recursion.
- General recursion permits a few operations that are not possible in surface recursion. If the reader plans to use those operations or simply wants to know everything about PL/I, then he should read the complete description of recursion.

Surface Recursion

A recursive program uses only surface recursion if all of the following statements are true:

- The program does not obtain the POINTER value that designates an AUTOMATIC variable during one activation of a block and then use that POINTER value to locate the AUTOMATIC variable during a later activation of the same block.
- The program does not obtain a statement address value (ENTRY, FORMAT, or LABEL value) during one activation of a block and then use the value to locate a statement during a later activation of the same block.
- The program does not establish an ON unit during one activation of a block and then signal the ON unit during a later activation of the same block.

In these statements, the phrase "activation of a block" refers either to the invocation of a procedure or the execution of a BEGIN block.

At certain points in the execution of a recursive program, a name must be interpreted as a reference to a storage unit in an activation region. However, the interpretation of a name given in the section on "Declarations" yields only the declaration of the name and the block in which its declaration is established. An additional rule is required to select one of the several activation regions for the given block. The general rule is complicated, and depends on extensions to the interpretation of several features of PL/I. However, the following special rule can be given:

> In a program that is restricted to surface recursion, any reference to an activation region for a given block uses the most recently created activation region that exists for that block.

Thus the interpretation of a program with surface recursion is scarcely more complicated than the interpretation of a nonrecursive program.

AN EXAMPLE OF RECURSION WITHOUT ARGUMENTS

The following program uses surface recursion to print a symmetrical list of integers:

```
R1:  PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL N FIXED;
        GET LIST(N);
        CALL SEQ;
SEQ:    PROC;
             DCL I FIXED;
             I = N;
             PUT SKIP LIST(I);
             IF N > 1 THEN
                  DO;
                  N = N-1;
                  CALL SEQ;
                  END;
             PUT SKIP LIST(I);
             END;
        END;
```

When this program is executed and the input value is 3, the output is:

```
3
2
1
1
2
3
```

Thus the program prints the integers from the given value down to 1 and then from 1 back up to the given value.

This program uses neither statement address variables nor ON statements, so it is immediately recognized as a case of surface recursion. The program is short and simple and therefore it is practical to describe its execution in detail, as follows:

1. Prepare for Program Execution. The external and static regions for the entire program are created. Two FILE constant storage units are allocated in the external region. Then R1 is invoked.

2. Start Execution of R1. The activation region for R1 is created, and storage for N is allocated in the region. The GET statement reads an input value (which is assumed to be 3) and assigns it to N. The CALL statement invokes SEQ, which is executed as follows:

   a. Start the First Execution of SEQ. An activation region is created for SEQ and storage for the automatic variable named I is allocated in it. The value of N is assigned to I and is printed as the first line of the output. Since N is greater than one, the value of N is reduced by one and then the CALL statement invokes SEQ, which is executed as follows:

1. Start the Second Execution of SEQ. A second activation region is created for SEQ and storage for the automatic variable named I is allocated in it. (Now there are two variables named I, and the rule for surface recursion must be applied to select one of them.) The value of N is assigned to I in the most recently created activation region for SEQ and is printed as the second line of output. Since N is still greater than one, the value of N is reduced by one and then the CALL statement invokes SEQ, which is executed as follows:

   A. Start the Third Execution of SEQ. A third activation region is created for SEQ and I is allocated in it. The value of N is assigned to I in the most recently created activation region for SEQ and is printed as the third line of the output. Since N is not greater than one, the reduction of N and the recursive call of SEQ are skipped. Now the recursive calls begin to "unwind".

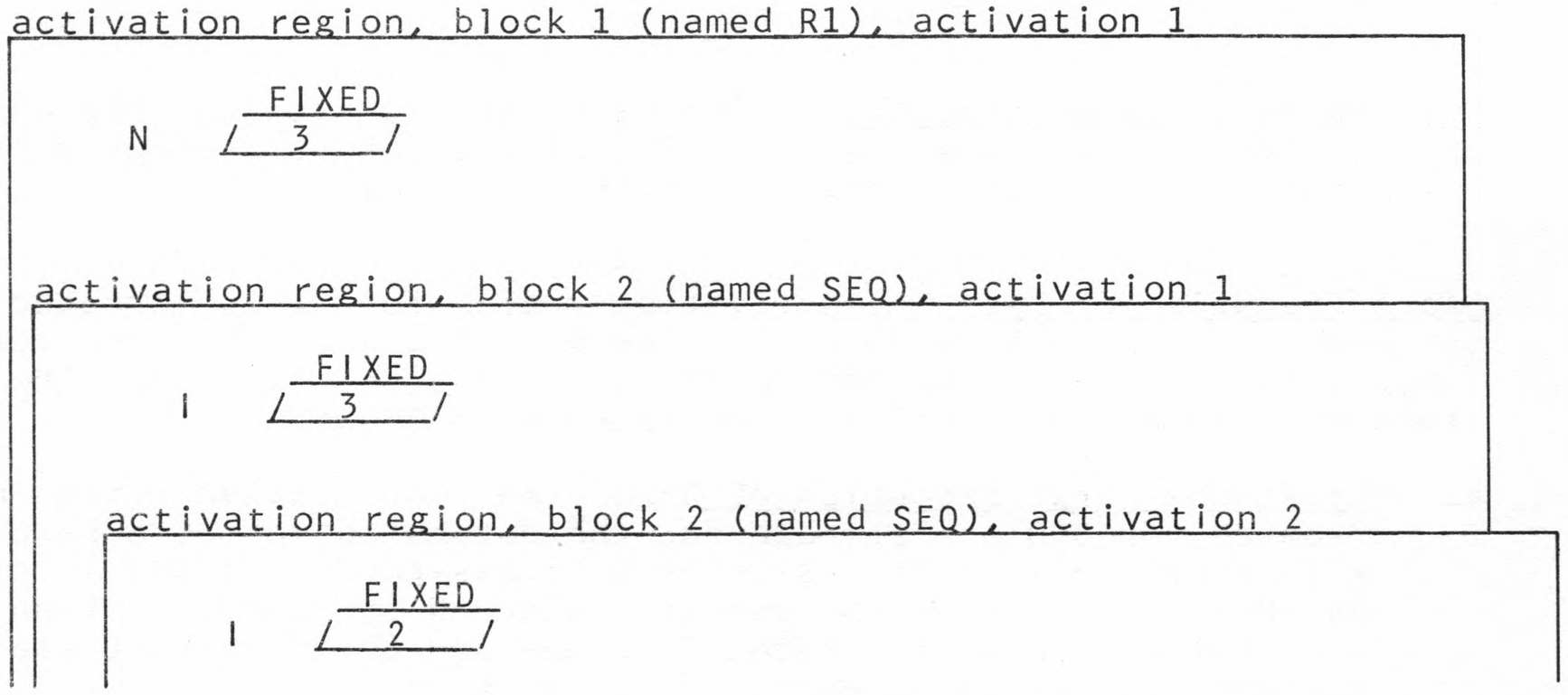
   B. Complete the Third Execution of SEQ. The value of I in the most recently created activation region for SEQ is printed as the fourth line of output. The third activation region for SEQ is discarded.

2. Complete the Second Execution of SEQ. The value of I in the most recently created activation region for SEQ is printed as the fifth line of output. (Of course "most recently created" refers only to those activation regions that still exist.) The second activation region for SEQ is discarded.

b. Complete the First Execution of SEQ. The value of I in the activation region for SEQ is printed as the sixth line of output. The activation region for SEQ is discarded.

3. Complete Execution of R1. The activation region for R1 is discarded.

In order to see how activation regions are handled during surface recursion, consider the following diagram of storage just before the start of the third execution of SEQ:



At this point in program execution, there are two activation regions for SEQ. According to the rule for surface recursion, a reference to I uses the one marked "activation 2".

AN EXAMPLE OF RECURSION WITH AN ARGUMENT

The preceding example program, named R1, can be simplified by transmitting an argument to SEQ. The new version is:

```
R2:  PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL N FIXED;
        GET LIST(N);
        CALL SEQ(N);
SEQ:    PROC(I);
           DCL I FIXED;
           PUT SKIP LIST(I);
           IF I > 1 THEN CALL SEQ(I-1);
           PUT SKIP LIST(I);
           END;
        END;
```

This program is "simplified" not only because it is shorter, but also because, for an experienced programmer, it shows more clearly how the procedure SEQ depends on the environment in which it is called.

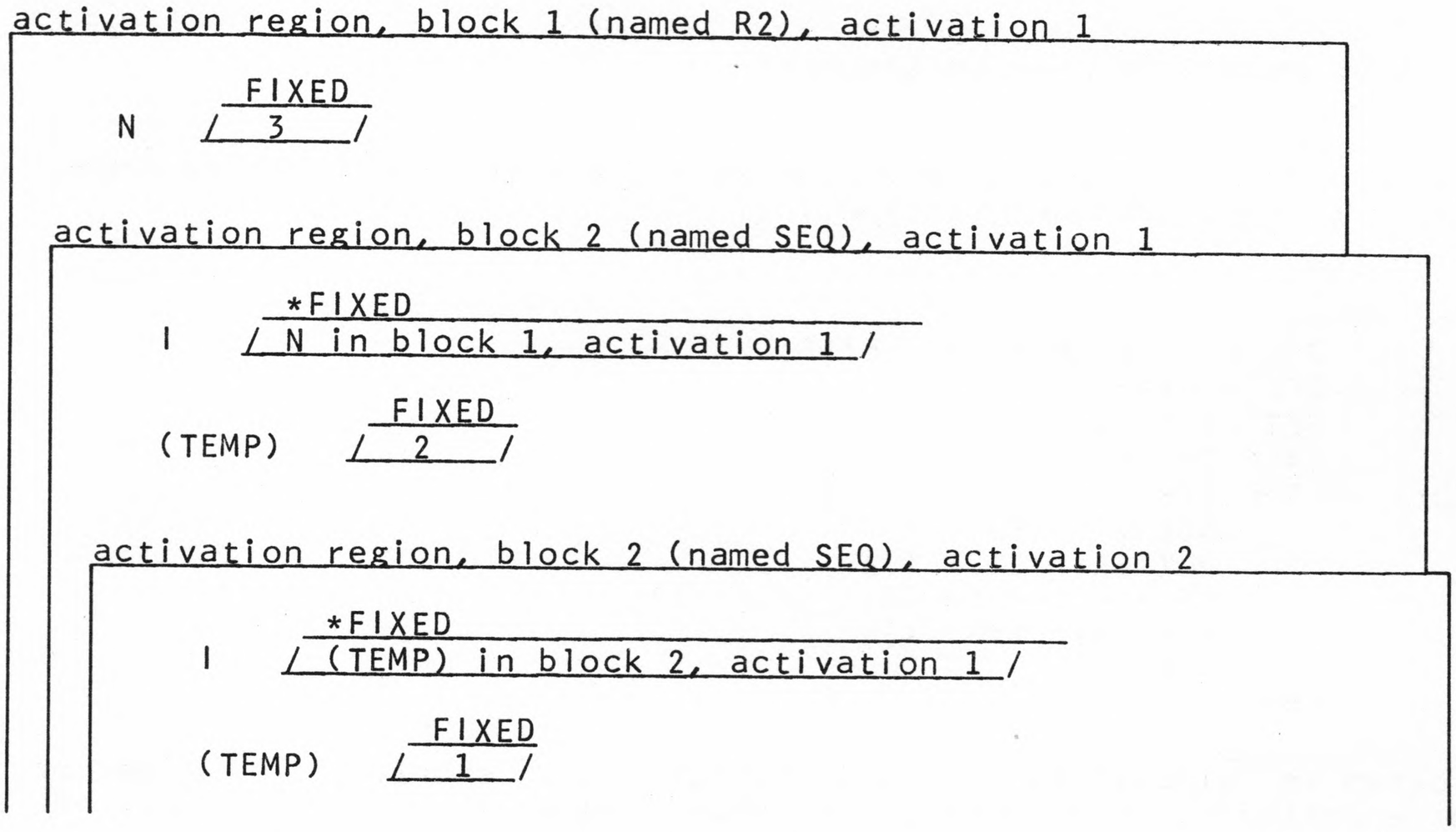
For each recursive invocation of SEQ, the program uses two system temporaries to transmit the argument. The temporaries play different roles, as follows:

- A FIXED temporary is used to hold the value of the argument I-1 in the second call. In GCOS PL/I, the argument temporary is allocated in the activation region of the calling procedure; therefore, the (n-1)th activation region contains the argument temporary for the nth invocation of SEQ.

- A POINTER temporary is used to link the parameter to the argument. A reference to the parameter I designates this parameter temporary, and the temporary, in turn, designates the storage unit for the argument. The parameter temporary is allocated in the activation region for the calling procedure; therefore, the (n-1)th activation region of SEQ contains the parameter temporary for the nth invocation of SEQ.

Observe that no argument temporary is required for the first call of SEQ because the argument N is transmitted by-reference, and the parameter temporary thus points directly to N.

In order to see how arguments are handled during surface recursion, consider the following diagram just before the start of the third execution of SEQ:



The storage unit for the parameter, I, is given the data type *FIXED to show that it holds a pointer to a FIXED storage unit. Because I is a parameter, a reference to I is interpreted as a reference to the storage unit designated by the pointer. The arrows in the diagram show how each parameter designates an argument storage unit. The storage unit designated (TEMP) is a system temporary used for the value of I-1 in the second CALL statement in the program.

AN EXAMPLE OF CHAINED RECURSION

For an example of chained recursion, consider once again the program R1 that was given as the first example of recursion. A program that produces the same output is:

```
R3:  PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL N FIXED;
        GET LIST(N);
        CALL SEQ;
SEQ:    PROC;
             DCL I FIXED;
             I = N;
             PUT SKIP LIST(I);
             CALL TEST;
             PUT SKIP LIST(I);
             END;
TEST:   PROC;
             IF N > 1 THEN
                  DO;
                  N = N-1;
                  CALL SEQ;
                  END;
             END;
        END;
```

There is no direct recursion in this version of the program: SEQ does not contain a call on itself and TEST does not contain a call on itself. However, SEQ calls TEST, and TEST calls SEQ, so recursion does occur.

A USEFUL RECURSIVE PROGRAM

The following program contains a recursive procedure, COMP, that is a rudimentary translator of expressions:

```
P:  PROC;
        DCL SYSPRINT FILE;
        CALL COMP("(((A-B)/C)*(D+(E+F)))",1,0);
        PUT SKIP;
COMP:   PROC(S,I,N) RECURSIVE;
            DCL S CHAR(*);
            DCL I FIXED;
            DCL N PIC"999";
            DCL A CHAR(9) VAR INITIAL("");
            DCL C CHAR(1);
            DCL K FIXED;
            DO K = I+1 BY 1;
                C = SUBSTR(S,K,1);
                IF C = "("
                    THEN DO;
                        CALL COMP(S,K,N);
                        A = A!!"T"!!N;
                        END;
                    ELSE IF C = ")"
                        THEN DO;
                            N = N+1;
                            PUT SKIP LIST("T"!!N!!"="!!A!!";");
                            I = K;
                            RETURN;
                            END;
                        ELSE A = A!!C;
                END;
            END;
        END;
```

The program is set up for a test run of the COMP procedure. The main procedure invokes COMP on an expression that is given as a character string; namely:

```
(((A-B)/C)*(D+(E+F)))
```

The output of the recursive execution of COMP is:

```
T001=A-B;
T002=T001/C;
T003=E+F;
T004=D+T003;
T005=T002*T004;
```

Thus each operator in the given expression is compiled into a PL/I statement.

The following assertions about variables help to explain the execution of the procedure:

S contains the parenthesized expression that is to be translated into a sequence of assignment statements. Its value does not change throughout the program. In fact, for each invocation of COMP, S designates the same storage unit; namely the temporary that is allocated for the argument of the first invocation.

I designates the '(' that begins the parenthesized expression within S that is to be translated by a particular invocation.

N contains the number of the most recently used name in the series T001, T002, and so on.

A contains the character string that represents the assignment statement that is being formed by the current invocation of the procedure. The variable is AUTOMATIC; otherwise the procedure would not work. For example, the initial invocation of the procedure proceeds as follows:

```
set A to ""
invoke COMP to compile "((A-B)/C)"
set A to "T002"
invoke COMP to compile "(D+(E+F))"
set A to "T002*T004"
```

Clearly the intermediate values of A would be lost if A were not allocated for each new invocation.

C contains the current character in the scan of the given expression. The variable is AUTOMATIC, but need not have been because there is no recursive call between its setting and its last use.

K designates the current character for a given execution of the body of the DO group; it controls the scan of the parenthesized expression. It is set to I+1 in order to skip the initial '('.

In order to study the procedure, simulate an invocation until the recursive call is reached. Instead of following that call, assume that it returns the correct result and complete the simulation of the current invocation.

General Recursion

Rules for the interpretation of general recursion are given here. The rules introduce extra steps into the execution of a program. These steps are essential if the program uses general recursion; otherwise, they have no effect on the final outcome. Thus the rules can be applied to all programs or just to those that use general recursion. In fact, the GCOS implementation of PL/I does not determine the recursive properties of a program but rather applies these rules to all programs. In contrast, a programmer who is studying a given program should probably ignore these rules unless the program uses general recursion.

General recursion requires special treatment of certain address values. The discussion that follows begins with the description of the activation index, which is used in an address value to designate an activation region. The discussion continues with a few paragraphs that explain the use of a POINTER value that has an activation index. The remainder of the discussion is devoted to statement address values: the parent index is introduced, rules for the use and setting of parent indexes are given, and several examples are included.

ACTIVATION INDEXES

A particular activation region is designated by adding an activation index to the designation for the block. Such an activation index is essential for a POINTER variable that designates an AUTOMATIC variable and for every statement address value. Consider, for example, the value described by "X in block 4, activation 2". Depending on the declaration of X, this address value is interpreted as follows:

- If X is an AUTOMATIC variable name, then the value is a POINTER value that designates the storage unit named X in the second activation of block 4.
- If X is a statement name, it designates the statement named X immediately contained in block 4 and, further, it specifies the interpretation of that statement within the second activation of block 4.

The way in which these values are generated and used is described later in this discussion of general recursion.

POINTERS

A POINTER value that points to storage in an activation region can be formed only by the application of the ADDR function to a reference to an AUTOMATIC variable. The argument of the ADDR function is interpreted to locate a specific variable in a specific activation region; then that information is expressed as a POINTER value. When general recursion is present, the POINTER value must include an activation index.

Once a POINTER value is formed, it can be assigned to a POINTER variable or returned as the result of a procedure. When, eventually, the POINTER value is used, it refers to the activation region that is specified by the activation index, even though that activation region may no longer be accessible in other ways. The following example illustrates this case.

An Example of Pointers

As an example of the formation and use of a POINTER value when general recursion is present, consider the following program:

```
Q1:  PROC OPTIONS(MAIN);
        DCL SYSPRINT FILE;
        DCL N FIXED STATIC INITIAL(1);
        DCL I FIXED;
        DCL IB BASED(IP);
        DCL IP POINTER STATIC;
        I = N;
        IF N = 1 THEN IP = ADDR(I);
        PUT SKIP LIST(I,IB);
        N = N+1;
        IF N < 3 THEN CALL Q1;
        END;
```

The output of this program is:

```
1          1
2          1
3          1
```

Each number on each line of this output represents the value of a variable named I. However, the first number in a line is taken from the variable that is allocated in the most recently created activation region, whereas the second number is taken from the variable that is allocated in the first (and least recently created) activation region.

The program is invoked three times, the first time by GCOS and the second and third times by itself, recursively. During the first, second, and third invocations, the value of N is 1, 2, and 3, respectively; and during each invocation, the current value of N is assigned to a variable named I that is allocated in the first, second, or third activation region. The important feature of the program is the statement:

```
IF N = 1 THEN IP = ADDR(I);
```

The assignment to IP is performed only during the first execution of the program and therefore the value of IP designates the same allocation of I throughout the program, namely "I in block 1 (named Q2), activation 1".

Suppose the IF statement is changed to the following:

```
IF N < 3 THEN IP = ADDR(I);
```

Then the output of the program is:

```
1       1
2       2
3       2
```

Suppose, on the other hand, that the IF statement is changed to the following:

```
IF N = 2 THEN IP = ADDR(I);
```

Then the program is invalid because a reference is made to IP (in the PUT statement, during the first invocation) before a value is assigned to IP (during the second invocation).

PARENT DESIGNATORS

In the presence of general recursion, each activation region must contain a parent designator. The parent designator in an activation region for a given block designates one of the activation regions for the block that immediately contains the given block. The parent designator in an activation region for an external procedure has no purpose and can be ignored. A parent designator is used in interpreting a program but is not directly accessible from the program.

The parent designators are used in the definition of a current activation region for each block that contains the most recently activated block. The definition is as follows:

- The current activation region for the most recently activated block is the most recently created activation region.
- The current activation region for a block that immediately contains a given block is the activation region designated by the parent designator of the given block.

As an example of the use of this definition, let the most recently activated block of a program be called B1; let the block that immediately contains B1 be called B2; let the block that immediately contains B2 be called B3; and let B3 be an external procedure. Then the current activation region for each of these blocks is determined as follows:

- The most recently created activation region for B1 is the current activation region for that block.
- Suppose that the parent designator in the current activation region for B1 designates the third activation region for B2; then the latter is the current activation region for B2.
- Suppose that the parent designator in the current activation region for B2 designates the first activation region for B3; then the latter is the current activation region for B3.

Thus the parent designators form a sequence of activation regions that begins with the most recently activated block and ends at the containing external procedure.

Observe that the blocks for which current activation regions are defined are exactly those blocks whose declared names can be referenced in the most recently activated block. The purpose of defining the current activation region is to resolve ambiguities that arise in the interpretation of certain references to those names. Such ambiguities arise in two cases: the interpretation of activation variable references and the evaluation of statement address constant names. Discussion of these cases follows.

Activation Variable References

Certain variables designate storage units that are allocated in an activation region. There are three such cases:

- An AUTOMATIC variable reference designates a variable that is allocated in an activation region.
- A DEFINED variable reference may or may not designate a variable that is allocated in an activation region; however, each extent (array bound, maximum string length, or area size) for the variable is contained in a system temporary that is allocated in an activation region.
- A PARAMETER variable reference designates a parameter pointer that is contained in a system temporary that is allocated in an activation region.

In each case, the designated storage is allocated in an activation region that is associated with the block in which the variable name is declared. If there is more than one such activation region, then the choice is resolved by selecting the current activation region for the block.

Statement Address Constant References

The evaluation of a given statement address constant reference yields an ENTRY value, a FORMAT value, or a LABEL value. Earlier in this discussion, it was noted that such values must include an activation index. That activation index is chosen so that it designates the current activation region for the block in which the name in the given reference is declared.

SETTING THE PARENT DESIGNATOR

An activation region is created when a PROCEDURE block, a BEGIN block, an ON unit, or a FORMAT statement is executed. For general recursion, the activation region must include a parent designator. Each of the four cases is described in the following paragraphs.

Part of the invocation of a procedure is the evaluation of an ENTRY reference. The resulting ENTRY value includes an activation region designator, and it is this designator that is used as the parent designator in the activation region that is created for the execution of the procedure.

The GCOS implementation of PL/I optimizes by avoiding the creation of a block activation region for certain procedures; however, the interpretation of the program is not changed. A programmer must be aware of this practice only when he examines stack frames directly. The compiler prints a message for each procedure that is compiled without an activation region.

As an example of procedure invocation in the presence of general recursion, consider the following program:

```
Q2:  PROC;
        DCL SYSPRINT FILE;
        DCL N FIXED STATIC INIT(0);
        DCL I FIXED;
        DCL EV ENTRY VARIABLE STATIC INTERNAL;
        N = N+1;
        I = N;
        IF N = 1 THEN EV = S;
        PUT SKIP LIST(I);
        CALL EV;
        IF N < 3 THEN CALL Q2;
S:      PROC;
           PUT LIST(I);
           END;
        END;
```

The output of this program is:

```
1          1
2          1
3          1
```

Each line of the output is produced by outputting the value of I twice; once by a statement in the main procedure and once by a statement in S. The first value of I is always from the most recent activation region for Q2, whereas the second value of I is from the first (least recent) activation of Q2.

The following aspects of the program just given are particularly interesting:

- A value is assigned to the ENTRY variable only once; namely, during the first activation of Q2. Therefore, the value of EV is equivalent to "S in block 1 (named Q2), activation 1".

- The procedure S is invoked three times, once in each activation of Q2. Each time, the ENTRY value has the activation index "activation 1", as just noted. Therefore, the parent designator in the activation region for S is always equivalent to "block 1, activation 1".

- Each time I is evaluated within S, the reference is to the current activation region for Q2. That activation region is specified by the parent pointer in the activation region for S, as just described, and is always activation 1.

The BEGIN Block

When a BEGIN block that is not an ON unit is executed, the parent designator in the activation region that is created for the execution of the block is set to designate the most recent activation region of the block that immediately contains the given BEGIN block.

The FORMAT Statement

The invocation of a FORMAT statement is similar to the invocation of a PROCEDURE block, but a FORMAT statement is used in a very restricted way, as described in the section on "Stream Input/Output". Part of the invocation of a FORMAT statement is the evaluation of a FORMAT reference. The resulting FORMAT value includes an activation region designator, and that designator is used as the parent designator in the activation region that is created for the execution of the procedure.

In order to reduce execution cost, the GCOS implementation of PL/I avoids the creation of a block activation region for a FORMAT statement. This optimization technique does not change the interpretation of a program, and it can be ignored by a programmer who remains within the framework of the PL/I language.

As an example of the invocation of a FORMAT statement, consider the following program:

```
Q3:  PROC;
        DCL SYSPRINT FILE;
        DCL N FIXED STATIC INIT(0);
        DCL I FIXED;
        DCL FV FORMAT VARIABLE STATIC;
        N = N+1;
        I = N;
        IF N = 1 THEN FV = F;
        PUT SKIP EDIT(2,2)(R(F),R(FV));
        IF N < 3 THEN CALL Q3;
F:      FORMAT(E(12,I));
        END;
```

The output of this program is

```
2.0E+000    2.0E+000
2.00E+000   2.0E+000
2.000E+000  2.0E+000
```

Each line of the output is produced by outputting the value 2 twice. Each output value is edited by the format statement F, but the format statement is invoked in two different ways and therefore the parameter I (which determines the number of digits after the decimal point) is evaluated in different activations. Specifically,

- The reference R(F) always causes the format statement to be evaluated within the most recent activation of Q3 so the value of I is 1, 2, and 3.

- The reference R(FV) causes the format statement to be evaluated within the activation specified in the FORMAT variable. Observe that the format variable is set only in the first activation of Q3. Therefore invocations of the FORMAT statement use the first activation region of Q3 and I is 1.

The ON Unit

When an ON statement is executed, it establishes (but does not execute) an ON unit for a given condition. As part of this action, the designation of the ON unit is associated with the given condition and saved. For the purposes of general recursion, the designation of the most recently created activation region for the block that immediately contains the ON statement is also associated with the condition and saved.

When a condition is signalled for which the program has established an ON unit, the most recently established ON unit is executed. As part of that execution, an activation region for the ON unit is created (regardless of whether the ON unit is a BEGIN block or a single statement). The parent pointer in that activation region is the activation region designator that was saved when the ON unit was established.

As an example of the invocation of an ON unit, consider the following program:

```
Q4:  PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL N FIXED STATIC INIT(0);
        DCL I FIXED;
        DCL K FIXED;
        DCL CONV COND;
        N = N+1;
        I = N;
        IF I = 1 THEN ON CONV PUT SKIP LIST("***",I);
        GET LIST(K);
        PUT LIST(I);
        IF I < 3 THEN CALL Q4;
        END;
```

Suppose the input for this program is:

```
6    5     FOUR
```

Then the output of the program is:

```
1
2
***       1
3
```

Since the ON condition is established during the first invocation of Q4, its associated activation index is "activation 1". During the third activation of Q4, the unacceptable input item, 'FOUR', is encountered, and the CONVERSION condition is signalled. The ON condition is invoked, but the parent designator in its activation region is equivalent to "block 1 (Q4), activation 1". For that reason, the ON unit prints out a 1.

Part of the execution of a GOTO statement is the evaluation of a LABEL reference. The resulting LABEL value includes an activation region designator. The GOTO statement causes exits from any block activations that are more recent than the block activation that is associated with the activation region designated by the LABEL value; then execution resumes at the designated statement.

As an example of the interpretation of a GOTO statement in the presence of general recursion, consider the following program:

```
Q5:  PROC;
        DCL SYSPRINT FILE;
        DCL N FIXED STATIC INIT(0);
        DCL I FIXED;
        DCL LV LABEL VARIABLE STATIC;
        N = N+1;
        I = N;
        IF I = 1 THEN LV = EXIT;
        IF I < 3 THEN CALL Q5;
        PUT SKIP LIST(I);
        GOTO LV;
EXIT:   END;
```

The output of this program is:

```
      3
```

The LABEL variable LV is set during the first invocation of the procedure to a value equivalent to "EXIT in block 1 (named Q5), activation 1". During activation 3 of the procedure, the GOTO statement is executed for the first time. The effect is an exit from activations 3 and 2 of Q5 and a transfer to the statement labelled EXIT.

As a rather different example of the interpretation of a GOTO statement, consider the following program:

```
Q6:  PROC;
        DCL SYSPRINT FILE;
        DCL N FIXED STATIC INIT(0);
        DCL I FIXED;
        DCL EV ENTRY VARIABLE STATIC INT;
        N = N+1;
        I = N;
        IF N = 1 THEN EV = S;
        IF N < 3 THEN CALL Q6;
        PUT SKIP LIST(I);
        CALL EV;
S:      PROC;
           GOTO EXIT;
           END;
EXIT:   END;
```

Once again the output of this program is a single line:

```
3
```

The statement address constant reference, EXIT, in the GOTO statement is evaluated during the third activation of the procedure Q6 in which EXIT is declared. However, its value is not "EXIT in block 1 (named Q6), activation 3". Instead, when S is invoked (for the first and last time), the parent designator in its activation region is "block 1 (named Q6), activation 1". Therefore, the current activation region of Q6 during the execution of S is the first activation region and the value of EXIT is "EXIT in block 1, activation 1". Just as in the previous example, the GOTO statement causes an exit from activations 3 and 2 of Q6 and a transfer to the statement labelled EXIT in activation 1.

SECTION XIII

CONDITION HANDLING

Sometimes a program instructs the processor to perform an action that cannot be performed. In this situation, it is said that an exceptional condition, or simply a condition, has occurred. In some cases, the occurrence of a condition is an error, as with an attempt to divide by zero. In other cases, the occurrence of a condition communicates the state of an external file, as with an attempt to read beyond the last record of a file. In all cases, some action must be specified as an alternative to the action that cannot be performed.

The PL/I processor can handle exceptional conditions by itself; in that case, it will, for most conditions, print an error message and abort the program. However, the PL/I language permits the programmer to supply a programmed response, called an ON unit, to the condition. In fact, the facilities for handling conditions are quite elaborate. These facilities are the subject of this section.

THE PRINCIPAL FEATURES OF CONDITION HANDLING

To handle a condition, the programmer first declares the condition by means of a DECLARE statement. Next, he enables the detection of the condition by means of a condition prefix unless the condition is already enabled. Finally, he establishes an ON unit for the condition by means of an ON statement. If the condition is signalled during the execution of the program, the established ON unit is executed.

As an example of condition handling, consider the following program:

```
P:    PROC;
         DCL ENDFILE COND;
         DCL (SYSIN,SYSPRINT) FILE;
         DCL X FLOAT;
         ON ENDFILE(SYSIN) GOTO EXIT;
LOOP:    GET LIST(X);
         PUT LIST(X,X**2);
         GOTO LOOP;
EXIT:    END;
```

This program illustrates the handling of the ENDFILE condition for the system input stream. First, the program declares ENDFILE as a condition name. The ENDFILE condition is always enabled, so the program does not enable it. Then the program establishes an ON unit for the condition reference ENDFILE(SYSIN). When the GET statement encounters an end-of-file, the ENDFILE condition is signalled for SYSIN and the established ON unit for that condition reference is executed. In this program, the established ON unit transfers control to the end of the program.

CONDITIONS

The condition handling facility of PL/I can be applied to two kinds of conditions, namely:

Language-defined conditions
Programmer-defined conditions

The language-defined conditions are defined as part of the PL/I language and detected by the PL/I processor during its execution of a program. The programmer-defined conditions are defined by the programmer and signalled by the program.

Language-Defined Conditions

The language-defined conditions are divided into four categories, as follows:

Computational
Storage
Termination
Input/Output

The following paragraphs define each category and then give a brief description of each condition in the category. The description ends with a reference to the section of this manual in which a complete description can be found.

COMPUTATIONAL CONDITIONS

The computational conditions occur when errors are detected during the conversion of values or the evaluation of expressions. The computational conditions are the only conditions that can be enabled and disabled. A description of the enabling and disabling of conditions is given later in this section.

The condition reference for a computational condition is the condition name or its abbreviation. The computational conditions are summarized in the following list.

Condition Reference	Description
CONVERSION CONV	occurs when an invalid character string or pictured value is converted to an arithmetic or bit-string value, as described in the section on "Conversion".
FIXEDOVERFLOW FOFL	occurs when the result of a binary fixed-point computation exceeds 71 digits, as described in the section on "Conversion".
OVERFLOW OFL	occurs when the result of a floating-point computation is too large to be represented, as described in the section on "Conversion".
SIZE	occurs when a value is assigned to a fixed-point target and the precision cannot accommodate the magnitude of the number, as described in the section on "Conversion".

STRINGRANGE STRG	occurs when a designated substring is not completely contained in the string argument of a SUBSTR function reference or pseudo-variable, as described in the section on "Operations".
STRINGSIZE STRZ	occurs when a character-string or bit-string is converted for a target whose length cannot accommodate the value, as described in the section on "Conversion".
SUBSCRIPTRANGE SUBRG	occurs when the value of a subscript exceeds the bounds of the dimension to which it corresponds as described in the section on "Expressions".
UNDERFLOW UFL	occurs when the result of a floating-point computation is too small to be represented, as described in the section on "Conversion".
ZERODIVIDE ZDIV	occurs when the divisor of a division operator is zero, as described in the section on "Expressions".

STORAGE CONDITIONS

The storage conditions occur when the capacity of PL/I storage is exceeded. In some cases, an ON unit can free storage and thus recover from a storage condition. If a storage condition is signalled and the established ON unit does not free sufficient storage, the condition is signalled again.

The condition reference for a storage condition is the condition name. The storage conditions are summarized in the following list.

Condition Reference	Description
AREA	occurs when an attempt is made to allocate storage in an area variable that cannot supply the storage, as described in the section on "Storage Management".
STORAGE	occurs when the stack segment is about to overflow or when the "system storage" in which CONTROLLED and BASED variables are allocated is full, as described in the section on "Storage Management".

TERMINATION CONDITIONS

The termination conditions occur when execution of a program is complete.

The condition reference for a termination condition is the condition name. The termination conditions are summarized in the following list.

Condition Reference	Description
FINISH	occurs when the main external procedure is terminated, as described later in this section.
ERROR	occurs as a result of a fatal error in program execution, as described later in this section.

INPUT/OUTPUT CONDITIONS

The input/output conditions occur when an input/output operation requires special attention. Some of the input/output conditions indicate errors, but other conditions communicate a valid change in the status of the data set.

The condition reference for an input/output condition consists of the condition name followed by a parenthesized file reference. The input/output conditions are summarized in the following list.

Condition Reference	Description
ENDFILE(fr)	occurs when an input statement attempts to read past the end-of-file on the referenced file, as described in the sections on "Stream Input/Output" and "Record Input/Output".
ENDPAGE(fr)	occurs when a PUT statement attempts to output a line that does not fit on the current page, as described in the section on "Stream Input/Output".
KEY(fr)	occurs when a KEY option specifies a key that does not exist in the referenced file or when a KEYFROM option specifies a key that already exists in the referenced file, as described in the section on "Record Input/Output".
NAME(fr)	occurs when an assignment is read from the referenced stream and the variable name does not match any of the variable names in the data list of the controlling GET statement, as described in the section on "Stream Input/Output".

RECORD(fr)	occurs when a READ statement reads a record from the referenced file that is not equal in size to the variable given in the INTO option, as described in the section on "Record Input/Output".
TRANSMIT(fr)	occurs when the data cannot be reliably transmitted between the referenced file and storage, as described in the sections on "Stream Input/Output" and "Record Input/Output".
UNDEFINEDFILE(fr)	occurs when an attempt to open the referenced file is unsuccessful, as described in the sections on "Stream Input/Output" and "Record Input/Output".

Programmer-Defined Conditions

The condition handling facility of PL/I can be used to define new conditions that are designed for the particular needs of a given programming application. By defining new conditions, a software system written in PL/I can provide its sub-systems with the capability of handling conditions.

CONDITION REFERENCES

In the discussion of conditions, a condition reference was given for each condition. The condition reference is used in the ON statement, the REVERT statement, and the SIGNAL statement to designate a condition. There are two kinds of condition references corresponding to the two kinds of conditions.

Language-Defined Condition References

A language-defined condition reference must have one of the following forms:

lcn

lcn(fr)

where lcn is a language-defined condition name and fr is the file reference. The language-defined condition name must be declared with the CONDITION attribute. For the first form the language-defined condition name must be one of the following identifiers:

AREA	FIXEDOVERFLOW	STORAGE	SUBSCRIPTRANGE
CONVERSION	FOFL	STRINGRANGE	SUBRG
CONV	OVERFLOW	STRG	UNDERFLOW
ERROR	OFL	STRINGSIZE	UFL
FINISH	SIZE	STRZ	ZERODIVIDE
			ZDIV

For the second form, the language-defined condition name must be one of the following identifiers:

ENDFILE	KEY	RECORD	UNDEFINEDFILE
ENDPAGE	NAME	TRANSMIT	UNDF

The file reference must be a reference that yields a scalar file value.

Programmer-Defined Condition References

A programmer-defined condition reference must have one of the following forms:

CONDITION(pcn) COND(pcn)

CONDITION pcn COND pcn

where pcn is the programmer-defined condition name. The programmer-defined condition name must be declared with the CONDITION attribute and must not be one of the identifiers listed as language-defined condition names in the preceding definition of language-defined condition references. The four forms of a programmer-defined condition reference are equivalent to one another.

THE DECLARATION OF CONDITION NAMES

Each condition name used in a program should be declared, as follows:

DCL cn COND;

where cn is the condition name. The condition name can be either a language-defined condition name or a programmer-defined condition name.

The only attribute that can be used with the CONDITION attribute is the scope attribute. In GCOS PL/I, every condition name must have EXTERNAL scope. Since this is the default scope for condition names, the scope attribute is omitted from the DECLARE statement.

ENABLING AND DISABLING CONDITIONS

The language-defined computational conditions can be enabled and disabled. When a condition that is enabled occurs, it is signalled and handled. When a condition that is disabled occurs, it may or may not be signalled.

The occurrence of a language-defined condition is detected either by the computer hardware or by additional code compiled into the program to test for the occurrence of the condition. When a condition that requires additional code for its detection is disabled, the resulting program requires less storage and executes more efficiently than a program for which the condition is enabled. However, if a disabled condition occurs in such a program, the program is invalid and the results of its continued execution are undefined.

Condition Prefixes

Condition prefixes are used to enable or disable language-defined computational conditions. The condition prefix must precede the label prefixes (if any) of a statement. The form of the condition prefix list is as follows:

(cpn, ...)

where cpn, ... is a sequence of condition prefix names separated by commas. The possible condition prefix names are enabling prefix names and disabling prefix names. An enabling prefix name is any of the computational condition names or their abbreviations. A disabling prefix name is the characters NO followed by an enabling prefix name. The complete list of condition prefix names is:

Enabling Prefix Names		Disabling Prefix Names	
Name	Abbr.	Name	Abbr.
CONVERSION	CONV	NOCONVERSION	NOCONV
FIXEDOVERFLOW	FOFL	NOFIXEDOVERFLOW	NOFOFL
OVERFLOW	OFL	NOOVERFLOW	NOOFL
SIZE		NOSIZE	
STRINGRANGE	STRG	NOSTRINGRANGE	NOSTRG
STRINGSIZE	STRZ	NOSTRINGSIZE	NOSTRZ
SUBSCRIPTRANGE	SUBRG	NOSUBSCRIPTRANGE	NOSUBRG
UNDERFLOW	UFL	NOUNDERFLOW	NOUFL
ZERODIVIDE	ZDIV	NOZERODIVIDE	NOZDIV

A condition prefix can begin any PL/I statement except the DECLARE, DEFAULT, or ENTRY statements.

The Scope of a Condition Prefix

A condition prefix applies to the statement to which it is attached. For most statements, the scope of the condition prefix is the entire statement; however, there are important exceptions, as follows:

Statement	Scope
PROCEDURE, BEGIN	The condition prefix applies to all statements in the block that begins with the PROCEDURE or BEGIN statement except those statements that lie within the scope of another condition prefix for the same condition.
IF	The condition prefix applies only to the test, which lies between the keywords IF and THEN; it does not apply to the consequences.
DO	The condition prefix applies only to the DO statement; it does not apply to the remaining statements of the DO group.
ON	The condition prefix applies only to the condition reference; it does not apply to the ON unit that follows the condition reference.
FORMAT	The condition prefix applies only to calculations performed in the evaluation of the format specification list; it does not apply to associated GET or PUT statements or to any other FORMAT statements.

As an example of the determination of the scope of a condition prefix, consider the following statement:

```
IF A(I,J) = 1
     THEN A(I,J) = B(K,M);
     ELSE A(I,J) = C(P,N);
```

To enable the SUBSCRIPTRANGE condition for the entire IF statement, it is necessary to apply the condition prefix to both the test and the consequences, as follows:

```
(SUBRG):
     IF A(I,J) = 1
          THEN
(SUBRG):       A(I,J) = B(K,M);
          ELSE
(SUBRG):       A(I,J) = C(P,N);
```

A different way of enabling the condition for the entire IF statement is to enclose the statement in a BEGIN block and then to apply the condition prefix to the block, as follows:

```
(SUBRG):
     BEGIN;
          IF A(I,J) = 1
               THEN A(I,J) = B(K,M);
               ELSE A(I,J) = C(P,N);
     END;
```

In practice, a condition prefix is usually applied to a procedure block and thus the condition is enabled or disabled for the entire procedure.

Default Enabling and Disabling

If a statement does not lie within the scope of a condition prefix for a given condition, then the statement is interpreted under the default enabling or disabling for the given condition, as follows:

Enabled by Default	Disabled by Default
CONVERSION	SIZE
FIXEDOVERFLOW	STRINGRANGE
OVERFLOW	STRINGSIZE
UNDERFLOW	SUBSCRIPTRANGE
ZERODIVIDE	

These defaults are dictated by considerations of efficiency. The conditions that are detected by the hardware are enabled by default and the conditions that require additional instructions in the compiled program for their detection are disabled by default.

An Example of Enabling and Disabling

As an example of enabling and disabling, consider the following program:

```
(SUBRG):
P:   PROC;
        DCL (A,B,C)(50,100) FIXED;
        DCL (I,J) FIXED;
        ...
        B(I,J) = A(I,J)**2;
(NOSUBRG):
        C(I,J) = B(I,J)**2;
(NOSUBRG,SIZE):
        C(J,I) = C(I,J)**2;
        ...
        END;
```

The SUBSCRIPTRANGE condition, which is disabled by default, is enabled for most of this program because of the '(SUBRG):' condition prefix on the PROCEDURE statement. However, SUBSCRIPTRANGE is disabled for the two assignments to the array C. Furthermore, the SIZE condition, which is disabled by default, is enabled for the last assignment statement.

ESTABLISHING AND REVERTING ON UNITS

A programmer has a choice between permitting the system to handle a signalled condition or writing a statement to handle the condition in a special way. The construct supplied by the programmer to handle the condition is an ON unit. An ON unit is established for a condition by the execution of an ON statement. When the condition is signalled, the established ON unit is executed.

In some cases, a single ON unit can be established to apply to all signals of a given condition. In other cases, the handling of the condition depends on the statement in which the condition occurs. In the second case, several ON units are established for a condition by the execution of several ON statements.

The establishment of an ON unit is associated with the current block activation. For a given block activation, at most one ON unit can be established at any time for each evaluated condition reference. Thus, for example, if an ON statement is executed for ENDFILE(SYSIN) and an ON unit for ENDFILE(SYSIN) is already established for the current block, the previously established ON unit is removed from the set of established ON units for the block before the current ON unit is added to the set. The execution of a REVERT statement removes an established ON unit from the set and re-establishes a previously established ON unit (or the system supplied ON unit if there was none previously established for the condition).

The rules for determining the established ON unit for a signalled condition are given later in this section in "The Occurrence and Signalling of Conditions".

The ON Statement

An ON statement has one of the following forms:

ON cr ou;

ON cr SNAP ou;

where cr is the condition reference and where ou is the ON unit. The ON unit must be one of the following:

- A BEGIN block; that is, a BEGIN statement, followed by a sequence of statements, followed by an END statement.
- A restricted independent statement; that is, one of the following statements:

storage management:	ALLOCATE and FREE
assignment:	the assignment statement
flow of control:	GOTO and the null statement
procedure invocation:	CALL
condition handling:	SIGNAL
input/output:	OPEN and CLOSE
stream input/output:	PUT and GET
record input/output:	READ, WRITE, DELETE, REWRITE and LOCATE

- The keyword SYSTEM

An ON unit can contain a RETURN statement only if the RETURN statement is contained in a PROCEDURE block that is contained in the ON unit.

When an ON statement is executed, the condition reference is evaluated and the ON unit for that evaluated condition reference is established. The evaluated condition reference and a designator for the ON unit are added to the set of established ON units for the current block activation. The ON unit in the ON statement is not executed until the condition is signalled. If the ON unit consists of the keyword SYSTEM, the default ON unit is established.

If the keyword SNAP is included in the ON statement, a special information list is written on the standard output file just prior to execution of the ON unit. The list gives the procedure names in the chain of procedures which are active at the time the ON unit is invoked and for which separate block activation regions exist. Also given with each procedure name in the list are the data types and values of the arguments used in the call to that procedure. The list begins with the system setup routine followed by the main program, it may include subprograms or system support routines invoked by the program object code, and it ends with the system condition handling routines followed by the ON unit.

The REVERT Statement

The REVERT statement has the following form:

REVERT cr, ...;

where cr is a condition reference.

When a REVERT statement is executed, the condition reference is evaluated and the ON unit for that evaluated condition reference is removed from the set of established ON units for the current block activation.

THE OCCURRENCE AND SIGNALLING OF CONDITIONS

A condition occurs as a result of the interpretation of the program by the PL/I processor. If a condition occurs when it is enabled, the condition is signalled and the established ON unit is executed. If a condition occurs when it is disabled, the program is invalid and the results of its continued execution are undefined.

If a SIGNAL statement for a condition is executed when the condition is enabled, the condition is signalled and the established ON unit executed. If the SIGNAL statement is executed when the condition is disabled, no action is taken and control passes to the next statement.

The Occurrence of Conditions

As an example of the occurrence of a condition, consider the following program:

```
P:   PROC;
        DCL (A,B,C) FIXED;
        A = 1;
        B = 5 - 5 * A;
        C = A / B;
        END;
```

The ZERODIVIDE condition occurs when the third assignment statement is interpreted by the PL/I processor. The ZERODIVIDE condition is enabled by default, so the condition is signalled. No ON unit is established for the condition, so the default ON unit is executed and the program halts with an error message.

The detailed description in this manual for each condition states when the condition can occur. Some conditions, however, are signalled from support subroutines when an error is detected or a limitation is exceeded. In these cases, the condition is signalled, even if it is disabled in the PL/I program, because the signal originates from a support subroutine in which the condition is enabled. The following conditions belong to this category:

```
AREA
ERROR
FIXEDOVERFLOW
OVERFLOW
SIZE
STORAGE
STRINGSIZE
ZERODIVIDE
```

These conditions are described as conditions that can occur at anytime during the execution of the program.

As an example of a condition that occurs at an unexpected time, consider the following program:

```
P:   PROC;
        DCL N FIXED(35) INIT(200000000);
        DCL X FIXED;
        DCL SYSPRINT FILE;
        X = N;
        PUT LIST(X);
        END;
```

The assignment of N to X is invalid since the value of N cannot be represented in the precision of X. The SIZE condition occurs on this assignment. Because the SIZE condition is disabled by default in the procedure P, the condition is not signalled. However, the SIZE condition is enabled in the run-time conversion routines and is signalled when the PUT statement is executed.

The SIGNAL Statement

The SIGNAL statement has the following form:

SIGNAL *cr*

where *cr* is a condition reference.

Execution of the SIGNAL statement signals the condition if it is enabled. If the condition is disabled, no action is taken.

Determining the Established ON Unit

As described earlier in this section under "Establishing and Reverting Conditions", each block activation has associated with it a set of established ON units. The set of established ON units for a block contains at most one established ON unit for an evaluated condition reference. When a condition is signalled, the established ON unit for that condition is determined in the following way:

1. Let the most recently activated block be the current block.

2. Examine the set of established ON units for the current block. If an established ON unit for the signalled condition is encountered, take that as the established ON unit.

3. If the current block is not the outer block, let the next most recently established block be the current block, and return to step 2. Otherwise, the program does not provide an ON unit, so take the default ON unit as the established ON unit.

Thus, the set of established ON units is searched, starting with the most recently activated block and proceeding back through the previously activated blocks.

THE ON UNIT

An ON unit specifies the action taken when a condition is signalled. The execution of an ON unit is similar to the execution of a procedure block. The ON unit is activated when the condition is signalled and, if control passes to the end of the ON unit, control is returned to the point at which the condition was signalled.

Some PL/I conditions are fatal conditions, namely:

```
AREA (if caused by assignment)
ERROR
FIXEDOVERFLOW
OVERFLOW
SIZE
STORAGE
STRINGRANGE
SUBSCRIPTRANGE
ZERODIVIDE
```

If an ON unit for any of these fatal conditions returns to the point at which the condition was signalled, the program is invalid and the results of its continued execution are undefined.

If an ON unit executed as a result of the signalling of a condition during the evaluation of an expression returns to the point at which the signal was detected, the ON unit must not allocate, free, or assign a value to any generation of storage accessible at the point where the condition was detected.

If an ON unit is executed as a result of the signalling of a condition during the execution of a statement, the ON unit must not access the value of the variable changed by the execution of the statement.

Default ON Units

If no ON unit is established by the program for a condition at the time the condition is signalled, the system default for the ON unit is invoked.

In GCOS, the following default ON units are defined for the listed conditions:

Condition	Default ON Unit
NAME UNDERFLOW	Writes an error message on the standard output file and returns control to the point at which the condition was detected.
STRINGSIZE	Returns control to the point at which the condition was detected.
ERROR	Writes an error message on the standard output file and halts.
FINISH	Closes any open files and returns to the point at which the condition was detected.
(other language-defined conditions)	Writes an error message on the standard output file and signals the ERROR condition.
(programmer-defined conditions)	Writes an error message on the standard output file and halts.

The ON Condition Built-In Functions

Seven built-in functions are associated with the condition handling capability of PL/I. The ON condition built-in functions are used to access system variables whose values are set by the system when a condition is signalled. These functions allow the programmer to obtain information for use in the ON unit that handles the condition. Some of the ON condition built-in functions can be used as pseudo-variables and, thus, the ON unit handling the condition can change some system variable value.

Each ON condition built-in function is associated with a stack. When a condition that sets the value of an ON condition built-in function is signalled, the old value of the function is pushed down on the stack and the new value is placed on the stack. When control returns to the block activion in which the condition was signalled or to any of the dynamic predecessors of the signalling block, the value is removed from the stack. The stack provides for the possibility that the execution of an ON unit for a condition causes a condition to occur.

The ON condition built-in functions are given in the following list. For each function, the value of the system variable is given. Also listed are the names of all the conditions whose occurrence alters the stack.

Built-In Function	System Variable Value	Associated Conditions
ONLOC	name of the most currently entered procedure block	all
ONCODE	error code value	all
ONCHAR	leftmost character for which the conversion failed	CONVERSION
ONFIELD	character string just extracted from the data stream	NAME
ONFILE	file name	CONVERSION, NAME, ENDFILE, TRANSMIT, RECORD, KEY, ENDPAGE, UNDEFINEDFILE
ONKEY	key value	ENDFILE, TRANSMIT, RECORD, KEY
ONSOURCE	string being converted	CONVERSION

The occurrence of the ENDPAGE condition, for example, provides a value for the ON condition built-in functions ONLOC, ONCODE, and ONFILE.

If the condition is signalled by the execution of a SIGNAL statement, the system variables have the values given in the following list:

Built-In Function	System Variable Value	Associated Conditions
ONCHAR	blank	CONVERSION
ONFIELD	null string	NAME
ONFILE	null string	CONVERSION, NAME, ENDFILE, TRANSMIT, RECORD, KEY, ENDPAGE, UNDEFINEDFILE
ONKEY	null string	KEY, ENDFILE, TRANSMIT, RECORD
ONSOURCE	null string	CONVERSION

The ONKEY built-in function is assigned a value by the SIGNAL statement for the conditions ENDFILE, TRANSMIT, and RECORD only if the file referenced has the keyed attribute.

A detailed description of the ON condition built-in functions is given earlier, in the section on "Operations". The ONCHAR and ONSOURCE functions can also be used as pseudo-variables. This use is described earlier, in the section on "Assignments".

AN EXAMPLE OF CONDITION HANDLING

As an example of condition handling, consider the following program:

```
P:    PROC;
         ON CONV
              BEGIN;
              ... (print warning message)
              ONSOURCE() = "Ø";
              END;
         ...
         CALL Q;
         ...
Q:       PROC;
              DCL X FLOAT;
              CALL R(X);
              PUT SKIP;
              PUT LIST(X,SIN(X));
              ON CONV
                   BEGIN;
                   ... (print warning message)
                   ONSOURCE() = "1";
                   END;
              CALL R(X);
              REVERT CONV;
              PUT LIST(X,1/X);
              CALL R(X);
              PUT SKIP LIST(X,X**2);
              END;
R:       PROC(Y);
              DCL Y FLOAT;
              GET LIST(Y);
              END;
         END;
```

The program P establishes an ON unit for the CONVERSION condition that prints a warning message and replaces the input string by the string "Ø". After some processing, the program P calls Q. The procedure Q calls the procedure R for input three times. The ON unit established by P is suitable for the first and third call on R, but for the second call on R a special ON unit is required.

The procedure R represents a general-purpose input program, which would, in practice, be more complex. The procedure R does not provide any handling for the CONVERSION condition because it cannot know the context of its call. The handling of the CONVERSION condition is entirely independent of the procedure in which it is signalled.

GUIDELINES FOR CONDITION HANDLING

The condition handling facility of PL/I is used both in debugging a program and in controlling the exceptional conditions that can occur during program execution. Guidelines for both applications are given here.

Debugging

During the debugging of a program, conditions that are normally disabled can be enabled and special ON units can be established.

ENABLING CONDITIONS FOR DEBUGGING

Four PL/I conditions provide additional error checking, namely; SIZE, STRINGSIZE, STRINGRANGE, and SUBSCRIPTRANGE. These conditions are normally disabled, since their detection requires the generation of additional code in the object program to perform the testing. For debugging, these conditions should be enabled by preceding each external procedure with the prefix:

```
(SIZE, STRZ, STRG, SUBRG):
```

When the program goes into production, the prefix should be removed.

ON UNITS FOR DEBUGGING

A useful debugging technique is the establishment of ON units to provide additional information about the state of the program when a condition is signalled.

Sometimes the same ON unit is established for every condition; namely, one that calls a debugging routine or produces standard information. Sometimes a different ON unit is established for each condition to produce debugging information specific to the condition. Sometimes, several ON units are established for the same condition, so the information produced depends upon the point at which the condition is signalled.

Controlling Exceptional Conditions

The condition handling facility of PL/I is used to control exceptional conditions. The file communication conditions are expected to occur and, consequently, ON units are usually established to handle these conditions. The error conditions, on the other hand, occur unexpectedly and the handling of these conditions is usually done by the default ON units provided by the system.

CONTROLLING FILE COMMUNICATION CONDITIONS

The ON unit established to handle a condition that communicates the status of an external file is part of the normal flow of a program. An example is a program that reads a file and uses an ON unit for the ENDFILE condition to transfer to the appropriate point in the program when the file is exhausted. A second example is a program that writes a report and uses an ON unit for the ENDPAGE condition to write a footing and heading on the report. A final example is a program that accesses a keyed file and uses an ON unit for the KEY condition to print a message and look in another file when the key is not found.

CONTROLLING ERROR CONDITIONS

Most programs do not establish ON units for the error conditions and if the condition is signalled, the default ON unit is executed. The default ON unit for most conditions prints an error message and terminates the execution of the program. In some cases, an alternative to the termination of the program can be defined. The following paragraphs consider these cases.

Input Data Validation

A program that reads input data often establishes an ON unit for the CONVERSION condition so that a bad input datum does not terminate the program. This ON unit can either report the bad input datum and read another or can attempt to correct the bad input.

Computational Checks

A program that is involved with computation often provides ON units for the OVERFLOW and UNDERFLOW conditions to change the course of an algorithm so that processing can continue.

Resource Management

A program that includes a system of storage management based on areas often provides an ON unit for the AREA condition.

Large Independent Systems

A large independent system or programming environment must handle all the language-defined conditions in order to maintain control over the processing. Moreover, such a system often makes use of programmer-defined conditions so that its users have the option of handling application-related conditions.

GENERAL CONDITIONS

The termination conditions are described in detail in this section since these conditions are related to the condition handling facility of PL/I and not to any particular PL/I language construct.

The ERROR Condition

The ERROR condition is signalled by the default ON units for several conditions, by the mathematical built-in functions, by the exponentiation operator, and by some run-time support routines.

The ERROR condition indicates a fatal error. If an ON unit for this condition attempts to return to the point at which the condition was signalled, the program is invalid and the results of its continued execution are undefined.

The default ON unit for the ERROR condition writes a comment on the standard output file and terminates the program.

The FINISH Condition

The FINISH condition is signalled by the execution of a statement that causes exit from the main external procedure. A RETURN or END statement terminating the execution of the main external procedure signals the FINISH condition.

The default ON unit for the FINISH condition closes any open files and returns to the point at which the condition was signalled. If termination resulted from the partial destruction of memory or exhaustion of resources, the FINISH condition is sometimes not signalled.

SECTION XIV

STREAM INPUT/OUTPUT

In PL/I, the object from which input values are taken or to which output values are transmitted is called a data set. There are two kinds of data sets, stream and record, and PL/I has a complete and independent input/output facility for each kind of data set. The facilities for stream input/output are oriented toward external media, such as line printers and card readers, and include elaborate features to assist the programmer in achieving a suitable format. In contrast, the facilities for record input/output are both more general and more primitive. This section describes the facilities for stream input/output, and the next section describes those for record input/output.

This section begins with a description of the two kinds of data that are involved in stream input/output: the stream, which is the actual subject of the input or output, and the file-state block, which shows the status of the operations on a stream. The section then gives a summary of the various operations that are performed as part of stream input/output. Once this foundation is established, the section proceeds to a definition of the statements that are used for stream input/output: first, the statements that open and close files, and then the statements that perform the actual input or output. Next, the section describes three different options for specifying the details of the format of stream input/output: data-directed, list-directed, and edit-directed. As the section nears its conclusion, a special feature of stream input/output, the string option, is presented. Finally, the section describes the conditions that occur in connection with stream input/output.

STREAM DATA SETS

The stream input/output statements operate on stream data sets or, more concisely, streams. A stream is a sequence of data characters and control characters. PL/I does not specify exactly what characters can be used as data characters, since this depends on the particular computer system and the input/output devices being used; however, any implementation of PL/I might be expected to have the letters, the digits, the common punctuation marks, and the blank among its data characters. There are two PL/I control characters, the linemark and the pagemark, and these represent the division between two lines or two pages, respectively.

A stream data set can be viewed as a character string that can be accessed only in special ways. During input, the characters of the stream are read in strict sequence, from left to right, and there is no way to return to a character that has already been read. During output, the characters of the stream are added at the right end of the stream, and there is no way to change a character that has already been written. The character string appears to flow past the PL/I interpreter and, for this reason, the data set is called a 'stream'.

The stream is a generalization of the various media that are used for communication between the user and the computer. It cannot handle diagrams or pictures, but it does capture the essence of the printed page or the punched card without introducing physical details of format, record boundaries, and so on. Furthermore, the processing of the stream is a good model for the behavior of hardware devices that are used for communication with the user. The single-pass, no-back-up property reflects the characteristics of line printers, card readers, and remote terminals.

The role of the stream is to make PL/I programs independent of the devices used for input and output. For example, a program can call for a sequence of one hundred arithmetic values without either telling a card reader to read another card or prompting an interactive terminal to supply another number. The program can call for input without knowing whether the input will come directly from a peripheral device, will be buffered, or will be waiting in permanent storage from some earlier input activity. Finally, by means that are discussed in this section, the program can deal with transmission errors and the end-of-file condition without reference to the specifics of the device that caused the condition to occur. Similar advantages apply to the process of stream output.

Control Characters in PL/I

PL/I assumes that the PL/I control characters, linemark and pagemark, are reserved for use in a stream and cannot be represented in a character-string value. In support of this assumption, PL/I provides statement options that are used to detect a linemark during input and generate a linemark or a pagemark during output. For example, to start a new page in the output stream, a program does not transmit a pagemark character; instead, it includes a PAGE option in the output statement, and the PAGE option causes a pagemark to be added to the output stream. This view of the control characters reflects the opinion that the starting of a new line or a new page is a rather special event in the composition of a printed document.

In GCOS PL/I, the stream data set is a sequence of ASCII characters. The ASCII 'new line' and 'new page' characters are used for linemark and pagemark, and most of the remaining ASCII characters are used for the PL/I data characters.

A difference in principle exists between the design of the PL/I stream data set on one hand, and the GCOS PL/I use of the ASCII character set on the other. PL/I views control characters as inseparably associated with the processes of input/output. GCOS PL/I does not normally restrict any characters in this way and views input/output as just one of many operations to which a character-string is subject. Indeed, the definition of the PL/I character-string value, given in the section on "Values", allows the use of any ASCII character in a character-string value. This difference in principle cannot be eliminated, but a practical solution can be achieved by asserting that

> It is an error to use a stream input/output statement to attempt to transmit an ASCII carriage return, new line, backspace, tab, or new page character between the stream and PL/I storage.

This restriction leaves latitude for other uses of the restricted control characters; specifically, it allows for the inclusion of these control characters in the string values transmitted by record input/output statements.

Input Streams

When a pagemark occurs in the input stream, it is treated as a blank; thus an input stream is treated as a single page divided into any number of lines. When the stream data set is opened, a stream pointer is associated with it and set to the first character. As the file is processed, this stream pointer proceeds from one data character to the next, advancing by however many characters are read or skipped. When the stream pointer reaches the end of the stream, the ENDFILE condition occurs.

For statements in which PL/I controls the editing of the input stream (that is, data-directed and list-directed input statements), the linemark has the effect of ending an item of input, and acts much as a blank does. This interpretation is in accord with the conventions of ordinary printed text, where an end-of-line can be used to separate two words.

For statements in which the programmer controls the editing of the input stream (that is, the edit-directed input statement), the linemark is ignored unless a specific reference to the line format is made. For example, if an input statement calls for three characters when the stream pointer selects the last character in a line, the resulting input will be the last character on the line and the first two characters on the next line; and no trace of the linemark will be input. On the other hand, an input statement can request that the stream pointer be moved to a certain column of a line or to the beginning of the next line.

There is no way to use stream input statements to program the operation "input the next line from the stream." When data-directed or list-directed input is used, the linemark is not distinguished from a blank. When edit-directed input is used, the programmer can skip the next line, but can perform input only by giving, in advance, the number of characters to be read. The operation in question can be performed by means of record input, which treats a line as a record.

Output Streams

When an output stream is opened, it is empty unless special arrangements are made to the contrary. As output is performed, characters are added at the right end of the stream. An output stream is usually intended for use to produce a printout. A PRINT output stream can contain both linemarks and pagemarks, and it thus exercises the full potential of the stream data set. A PRINT stream should (within the PL/I framework) be used only for printout; that is, the stream should not be used later as a PL/I input stream.

Generally speaking, PL/I allows a programmer to control the format of the output and intervenes only when the programmer neglects this activity. Specifically, the maximum length of a line and the maximum number of lines on a page are established when a stream data set is opened for output. If a program attempts to write beyond the end of a line, PL/I automatically inserts a linemark and forces the beginning of a new line. Similarly, if a program attempts to write beyond the end of a page, the ENDPAGE condition occurs. The programmer can establish an ON unit for ENDPAGE to start a new page and provide a suitable heading, or else he can allow PL/I to start a new page as the default response to the ENDPAGE condition.

It is possible to use a stream data set as permanent storage. That is, a sequence of values can be written as output to a stream data set, left until they are needed, and then read as input from the same data set. However, this use of a stream data set is not recommended. It is difficult to imagine a case in which a record data set would not be a more efficient, simple, and accurate means for permanent storage.

Pseudo-Streams

A character-string storage unit can be used as if it were a stream, and in this role, it is called a pseudo-stream. For an input statement, characters are taken from the string variable just as if it were an input stream. For an output statement, the value of the string variable is first initialized to null and then characters are added to its value just as if it were an output stream. Under these circumstances, no actual input or output occurs, but the large and complicated editing facility of the stream input/output statements can be applied to the editing of character strings.

A pseudo-stream cannot contain either a linemark or a pagemark, and thus is viewed as a single line of data characters. This restriction is consistent with the PL/I requirements that control characters can only be used in (true) stream data sets.

STREAM INPUT/OUTPUT FILES

A connection must be established between a statement that performs input/output and the GCOS file on which the operation is to be performed. A detailed analysis of this connection follows:

- The connection begins with an input/output statement.
- Every input/output statement contains a file option.
- A file option has as its argument a file reference.
- The evaluation of a file reference yields a file value.
- A file value is a pointer-like object that designates a file-state block.
- A file-state block is a structure-like set of values that includes a GCOS file code.
- A file code does, indeed, designate a file and is thus the last part of the connection.

The PL/I data sets have already been discussed in this section, and the input/output statements will be discussed later. For the present, the subject is the file-state block and the file reference that designate it.

File-State Blocks

Transmission of values between PL/I and a GCOS file requires bookkeeping data. This data is stored in a file-state block. When a file is open, the file-state block contains the GCOS file code and other information about input/output in progress. After the file is closed, the only information in the file-state block that is meaningful is that supplied by the attributes, if any, in the declaration of the file constant name. Although a file-state block resembles a structure, its values cannot be accessed directly by a PL/I program; instead, it is used and maintained by the PL/I processor.

The following values in the file-state block are relevant to stream input/output:

- The status indicator, which shows whether the file is 'open' or 'closed'; that is, whether or not a data set is currently attached to the file
- The GCOS file code, which is used to establish the actual connection between the file-state block and a file
- The file name, which is the file constant name (an identifier) expressed as a character-string value
- The file attributes associated with the current use of the file-state block. These are discussed later; one of them shows whether the file is an input file or an output file
- For an input file, the stream pointer, which points to the character which will be read next
- For an output file, the line size and page size, which give maximums for the number of characters per line and the number of lines per page, respectively
- For an output file, the column position, the line number, and the page number, which have values i, j, and k if the next character output will be the (i+1)th character in the jth line on the kth page

It is customary to use the word "file" as an abbreviation for the term "file-state block", and some other liberties are taken to attain brevity. For example, one might say "advance the stream pointer associated with the file TEST2 through the input stream," instead of saying "advance the stream pointer contained in the file-state block associated with the file value designated by the file constant TEST2 through the stream data set associated with the same file-state block." No confusion arises if one remembers that a remark about the actual data refers to a data set, while a remark about the control of the transmission process refers to a PL/I file-state block.

File References

A file-state block is designated by a file value; and the file value is supplied to an input/output statement by a file constant, a file variable, or a file-valued function. For example, the input statement

```
GET FILE(TEST2) LIST(A,B,C);
```

uses the file constant TEST2 to refer to a file-state block, which, in turn, refers to a file.

A file constant should be declared with the following attributes:

```
[EXTERNAL]
[INTERNAL]   FILE   [CONSTANT]
```

When the scope attribute is EXTERNAL, it can be omitted. The CONSTANT attribute can always be omitted; however, some programmers prefer to write FILE CONSTANT in a declaration in order to avoid confusion with a file variable.

Every file constant name must have an associated file description. This file description may be given when the file is opened, as described later in this section, under "The OPEN Statement". However, PL/I also allows the programmer to write any portion of the file description attributes in the declaration of the file constant name.

For each declaration of a file constant, a file-state block exists in static storage. The only exception is the declaration of a given identifier as EXTERNAL FILE CONSTANT in several places; in this case, the declarations refer to the same file-state block, as required by the proper interpretation of the EXTERNAL attribute. A given file constant and its associated file-state block can be used for more than one data set in the course of a PL/I program execution. For example, a file-state block can be opened for input from a stream data set, closed, opened for output to a record data set, and closed again.

A file variable or file function is declared similarly to an arithmetic variable or function. The only differences are:

- The data type is FILE.
- The default scope EXTERNAL applies to a file variable, whereas the default scope for most other data types is INTERNAL.
- The attribute VARIABLE must be used explicitly for a file variable because PL/I will otherwise assume the identifier is a FILE CONSTANT.

STREAM INPUT/OUTPUT OPERATIONS

The summary of stream input/output operations that follows includes terminology, shows how stream data sets are manipulated, and gives a general view of the stream input/output facility. It also serves as an introduction for the subsequent pages of this section.

When a file is opened for input the designated file is attached and preparations are made for reading from the first character onward. When a file is opened for output, the previous contents of the designated file are discarded and the file is prepared for the appropriate output format. A stream file can be opened for input or for output, but not for a combination of input and output.

Input operations proceed through the file reading or skipping characters in strict sequential order. Output operations write characters, always adding them at the right end of the stream. Thus the actual transmission of data is simple. The complexity of stream input/output arises from the variety of ways in which the GET statement edits the characters that are read from the input stream and the equally numerous ways in which the PUT statement edits the characters that are added to the output stream.

Within stream input/output, there are three separate disciplines of input/output. The first two disciplines, data-directed and list-directed, are closely related and are both quite automatic; that is, PL/I makes most of the decisions about the representation of values and the layout of a page. The third discipline, edit-directed, allows the programmer to specify the details of representation and layout. Within this last discipline there is yet a further choice of methods. The programmer can choose between format items, which are derived from FORTRAN, or pictures, which are derived from COBOL. Clearly, the choice of an input/output discipline can be difficult, and advice on the choice will be given.

Many conditions can occur as the direct or indirect result of stream input/output. Certain conditions always imply that an error has occurred; other conditions are used to control the logic of the input/output process and do not necessarily indicate an error. Some conditions are uniquely associated with input/output; other conditions arise from common operations, such as the assignment of a value to a variable, that happen to be used during input/output.

This concludes the summary of stream input/output operations. The remainder of this section is devoted to the detailed consideration of these operations.

OPENING AND CLOSING FILES

When a file is opened, the file-state block is marked "open" and the data set designator and control parameters and indexes are set in the block. When a file is closed, the file is marked "closed" and only information provided by the file declaration is meaningful.

A file is opened when the first input/output statement referencing the file is executed. The purpose of the OPEN statement is to provide the title and file description for the file opening. However, both these options can be omitted from the OPEN statement, and, in that case, a default assumption is made. If an OPEN statement is not given for a file, the attributes for the file opening are derived from the first input/output statement executed. If a file is already open when an OPEN statement is executed, the OPEN statement is completely ignored.

The OPEN Statement

When an OPEN statement is used to open a stream for input, it gives a file value, a title for the stream data set, and a file description. Consider the statement:

```
OPEN FILE(TEST2) TITLE("Z1") INPUT;
```

In this statement, the file value is given by the constant TEST2, the title is Z1, and the file description is INPUT. This statement is interpreted as follows:

- The title is used to determine the file code.
- The associated data set is checked to make sure that it is a stream data set.
- The column position associated with the file is set to 1, and the stream pointer is set to point to the first character of the stream data set.
- The file is marked "open".

If any of these steps cannot be performed, the UNDEFINEDFILE(TEST2) condition occurs. If the file designated by TEST2 is already open when the statement is executed, then the OPEN statement is ignored.

When an OPEN statement is used to open a stream for print output, the maximums for the length of a line and a page can also be given. Consider the statement:

```
OPEN FILE(REPORT) TITLE("ALPHA") PRINT OUTPUT
     LINESIZE(80) PAGESIZE(60);
```

The statement is interpreted as follows:

- The property page size associated with the file is set to the value given by the PAGESIZE option, namely 60.
- The property line size is set to the value given by the LINESIZE option, namely 80.
- The title is used to determine the file code as in the previous OPEN statement.
- The data set is deleted and a new data set conforming to the file description STREAM PRINT OUTPUT is created.
- The indexes line number, and page number are set to 1 and the index column position is set to 0.
- The file is marked "open".

As in the previous example, the UNDEFINEDFILE(REPORT) condition occurs if any of these steps cannot be performed, and the OPEN statement is ignored if REPORT is already open.

If the title option is omitted from an OPEN statement, the file name is used in forming a title. The statement

```
OPEN FILE(REPORT) PRINT;
```

is equivalent to

```
OPEN FILE(REPORT) TITLE("REPORT") STREAM PRINT OUTPUT ENVIRONMENT(CONSECUTIVE)
                LINESIZE(132) PAGESIZE(55);
```

This example also shows that the default maximum for the length of a line is 132 characters, the default maximum for the number of lines on a page is 55, and PRINT implies an output stream.

There are not many file descriptions for stream input/output. The important ones were used in the example of the OPEN statement, above; they are

```
{ INPUT          } [ ENVIRONMENT ( { CONSECUTIVE  } ) ]
{ PRINT [OUTPUT] }                 { INTERACTIVE }
```

The PRINT attribute should not be used in opening an output stream that, at a later time, will be used as an input stream. When an output stream is directed at an interactive terminal, the ENVIRONMENT(INTERACTIVE) attribute may be used.

The CLOSE Statement

The CLOSE statement has a simple form, as indicated by the following example:

```
CLOSE FILE(TEST2);
```

This statement marks the file-state block TEST2 closed. If the program is interrupted before a file has been closed, its contents are undefined. Therefore, every OPEN statement should be matched by a CLOSE statement; and, further, the CLOSE statement should be executed as soon as possible after the completion of input/output operations on the file.

Default Files

It is possible to write a PL/I program without declaring a single file constant or using a single OPEN statement. In that case, input is taken from the system input file, and output goes to the system output file. Consider first the case of a stream input statement that does not have a file option; for example,

```
GET LIST(A,B,C);
```

The default mechanism of PL/I comes into play with full force here. Before execution of the program begins, PL/I inserts an OPEN statement and adds a file option to the GET statement, giving

```
OPEN FILE(SYSIN) TITLE("I*") INPUT;
GET FILE(SYSIN) LIST(A,B,C);
```

If the statement does not lie within the scope of a declaration of SYSIN, PL/I supplies the following declaration in the largest enclosing block:

```
DCL SYSIN EXTERNAL FILE CONSTANT;
```

The omission of an OPEN statement is common. On the other hand, SYSIN and SYSPRINT should always be declared because GCOS PL/I requires that every name should be explicitly declared.

The default interpretation just described is applied to every GET statement in a program that does not have a FILE option. Because SYSIN is declared EXTERNAL all such statements refer to the same file-state block, even when the interpretation of several GET statements leads to several declarations of SYSIN. Because an OPEN statement only performs an action when the designated file is not already open, the file SYSIN is only opened once.

Consider next a stream output statement that does not have a file option; for example,

```
PUT LIST(X,Y,Z);
```

This statement is treated like the input statement, except that SYSPRINT is used as the default file constant. The statement is expanded to be:

```
OPEN FILE(SYSPRINT) TITLE("P*") PRINT;
PUT FILE(SYSPRINT) LIST(X,Y,Z);
```

INPUT/OUTPUT STATEMENTS

There is one statement for stream input, the GET statement, and one statement for stream output, the PUT statement. Each statement is a keyword followed by a sequence of options. Most of the options are simple. They specify the file on which the statement will operate, the number of lines to be skipped before reading or writing begins, and so on; and these simple options are described here. The final option in a GET or PUT statement is the transmission option and it is not simple. It specifies the transmission of data according to the rules of data-directed, list-directed, or edit-directed input/output. The transmission option is described later in this section.

In addition to the GET and PUT statements, there is a third stream input/output statement, the FORMAT statement. This statement plays a specialized role, and is used only in connection with an edit-directed transmission option.

The GET Statement

Stream input is performed by the GET statement. A full example of such a statement follows:

```
GET FILE(TEST2) COPY(ECHO9) SKIP(N+2) LIST(A,B,C);
```

There are four options in this example. The options are interpreted as follows:

- FILE(TEST2) is the file option. It designates a file-state block and, through the file-state block, the stream data set from which input is to be taken. If the option is omitted, FILE(SYSIN) is assumed.

- COPY(ECHO9) is the copy option. When this option appears, every character skipped or read from the input stream is written in the output stream designated by this option. If a COPY is given without an argument, COPY(SYSPRINT) is assumed.

- SKIP(N+2) is the skip option. It specifies that, before any input is performed, characters will be skipped until the beginning of the (N+2)th line after the current line. If SKIP is used without a count, SKIP(1) is assumed.

- LIST(A,B,C) is the transmission option. In this case it is list-directed and specifies that the next three values in the stream will be read in and assigned to the three targets given in the list, namely: A,B, and C. If the option is omitted, no values are input.

The example just given shows the important input options. The following points round out the description of the GET statement:

- The skip and transmission options cannot both be omitted, since the statement would then do nothing; but options can be omitted in any other way. In fact, the use of all options, as in this example, is rare.

- The options can be arranged in any order, but the order used in this example is recommended because it is the order in which the actions are performed. The source and copy files are prepared for transmission, then the skip is performed, and only then does input from the stream occur.

- It is possible to simulate input by using the option STRING(e) instead of the file option, where e is a character-string expression. The value of e is treated just as if it were the current input stream. More is said of this later, under "The String Option".

The PUT Statement

Stream output is performed by the PUT statement. A full example of such a statement follows:

```
PUT FILE(REPORT) PAGE LINE(5) LIST(X,Y,Z);
```

There are four options in this example. The options are interpreted as follows:

- FILE(REPORT) is the file option. It designates the stream data set to which the output will be transmitted. If the option is omitted, FILE(SYSPRINT) is assumed.

- PAGE specifies that a pagemark will be written in the output stream.

- LINE(5) specifies that sufficient linemarks will be written so that subsequent output will begin the fifth line of a page. If a new page must be started, the subsequent output begins on the first line of the new page.

- LIST(X,Y,Z) is the transmission option. In this case it is list-directed and specifies the values that are to be written out. If the option is omitted, no values are output.

The example just given does not illustrate the following features of the PUT statement:

- A SKIP(n) option can be used in a PUT statement. It specifies that subsequent output should begin the nth line after the current line. If SKIP is used without an argument, SKIP(1) is assumed. If the skip option is used, neither the page nor line option can be used.

- The skip, page, line, and transmission options cannot all be omitted, since the statement would do nothing; but options can be omitted in any other way.

- The options can be arranged in any order, but the order used in the example is recommended because it is the order in which actions are performed: The destination file is located, then the page and line options are performed (in that order), and only then does output to the stream occur. If a skip option is used, it should be written just after the file option so that it takes the place of any page or line option.

- It is possible to simulate output by using STRING(t) where t is a suitable target for assignment of a character-string value. The string of characters that would otherwise be placed in the output stream is assigned to t.

The FORMAT Statement

The FORMAT statement is the keyword FORMAT followed by a parenthesized format list. The statement must begin with at least one label prefix. Consider the statement

```
F5:  FORMAT(A(10),P"BBB--9V.99");
```

This statement has the label prefix 'F5:' and the format list is made up of the items A(10) and P"BBB--9V.99".

The purpose of a format statement is to supply a format list for an edit-directed GET or PUT statement that appears elsewhere in the program. Specifically, such a GET or PUT statement may contain a remote format item, such as

```
R(F5)
```

The remote format item is interpreted by interpreting the items in the format list associated with F5; namely,

```
A(10), P"BBB--9V.99"
```

Thus the FORMAT statement is used in order to supply a format list for some other stream input/output statement. The interpretation of the format list itself will be given later, in the discussion of edit-directed input/output.

A FORMAT statement is similar in some respects to a procedure. The identifier F5 is a format constant name, not a label constant name. It is used to invoke the FORMAT statement only in a remote format item, and it cannot be used as the destination of a transfer of control. When control reaches a format statement as a result of the sequential execution of the preceding statements, the format statement is skipped, just as a procedure is skipped under the same circumstances.

The value of a format constant is similar in many respects to an entry value. The format value can be assigned to a format variable, can be the result of a reference to a format variable or a function, and can be compared to another format value by means of the operators '=' and '^='. A variable or function that has a format value is declared similarly to an arithmetic variable or function; its data type is FORMAT, and the attribute VARIABLE is always assumed to apply.

As an example of the use of a format variable, and as a further example of the format statement, consider the following program fragment:

```
     ...
     DCL X FORMAT;
Q:   FORMAT(A(10),F(11,2));
     ...
     X = Q;
     ...
GET EDIT(Y, Z)(R(X));
...
```

The remote format item has the format variable X as its argument. Since the value of X is the format constant Q, the remote format item is equivalent to R(Q). Furthermore, since the format constant Q is associated with the format list (A(10), F(11,2)), the GET statement is equivalent to

```
GET EDIT (Y, Z)(A(10), F(11,2));
```

The interpretation for this kind of input statement is given later; the purpose here has been to show how format constants, variables, and statements can be used to supply a format list to such a statement.

DATA-DIRECTED INPUT/OUTPUT

When data-directed input is used, the input stream contains a variable for each input value; so the target for an input value is provided by the data rather than by the program. Furthermore, once a data-directed statement has initiated input, the process continues until a semicolon is encountered in the input stream; so input is terminated by the data rather than the program. For these reasons, the term "data-directed" is applied to this input/output facility.

An Example of Data-Directed Input/Output

As an example of data-directed input/output, consider the following program to calculate the range of an artillery piece fired on level ground:

```
RANGE:  PROC;
            DCL (SYSIN,SYSPRINT) FILE;
            DCL SIND BUILTIN;
            DCL (V0, THETA, RANGE) FLOAT(15);
            DCL G FLOAT(15) INIT(32.174);
            DO WHILE("1"B);
                GET DATA(V0, THETA);
                IF V0=0 THEN RETURN;
                RANGE = ((V0**2)*SIND(2*THETA))/G;
                PUT SKIP DATA(V0, THETA, RANGE);
                PUT SKIP;
                END;
            END;
```

This program appears to be an infinite loop (since the condition WHILE("1"B) is always satisfied). Each time around the loop, the program reads the muzzle velocity (V0) and the angle of elevation (THETA), performs an end test, calculates the distance to impact (RANGE), and outputs results. The program is designed to stop on a misfire; that is, it returns when V0=0.

The input statement in this program is the data-directed GET statement

```
GET DATA(V0,THETA);
```

Each time this statement is executed, the input stream is read through the next semicolon in the stream. Suppose the program is being used to calculate four trajectories. Then the input stream might be:

```
V0=1000   THETA=35;
V0=1000   THETA=40;
V0=1000   THETA=45;
V0=1280   THETA=45;
V0=0      THETA=45;
```

In this example, each line ends with a semicolon and therefore represents the two values required for an execution of the GET statement and the calculation of a trajectory. The zero value of V0 stops the program, but the last value of THETA is added for completeness and has no effect whatever.

A value not mentioned in the input stream is not changed; furthermore, the order in which values are mentioned has no significance. Therefore, the same trajectories can be specified as follows:

```
THETA=35     V0=1000;
THETA=40;
THETA=45;
V0=1280;
V0=0;
```

The stream assignments can be separated from one another by any sequence of blanks, tab characters, and new lines; and thus the input data can be attractively formatted (as above) and can be broken into several lines when it does not fit on one line.

It is the semicolon and not any other character that delimits the input stream read by a given GET statement. Thus the example can be written in yet another way, this time as a compact single line:

```
THETA=35 V0=1000;THETA=40;THETA=45;V0=1280;V0=0;
```

This format is less attractive but, as a practical matter, it might well be used when the input is being typed in at an interactive terminal.

Why are the variables V0 and THETA mentioned in the GET statement? Since the input stream specifies a variable for each input value, a mention of variables in the GET statement appears to be redundant -- and it is. The effect of the mention of variables in the GET DATA statement is to restrict the input stream to those variables only. Thus, for example, if the input stream given above included G=1000, PL/I would reject the input and cause the NAME condition to occur because the GET DATA statement does not mention G. This restriction allows the programmer to maintain control over the effects of data-directed input and also allows PL/I to execute data-directed input more efficiently.

The output statement in the RANGE program is the data-directed PUT statement

```
PUT SKIP DATA(V0, THETA, RANGE);
```

If the input stream is the one discussed above, the program executes this PUT statement four times and produces the following addition to the output stream:

```
V0= 1.0000E+003     THETA= 3.5000E+001  RANGE= 2.9207E+004;
V0= 1.0000E+003     THETA= 4.0000E+001  RANGE= 3.0609E+004;
V0= 1.0000E+003     THETA= 4.5000E+001  RANGE= 3.1081E+004;
V0= 1.2800E+003     THETA= 4.5000E+001  RANGE= 5.0923E+004;
```

The preceding output is directed to the SYSPRINT file, which is declared PRINT. For a file with the PRINT attribute, PL/I assumes the tab stops of the printer are set at 11, 21, 31, and so on. Each stream assignment (except the first) is preceded by a tab character, and each stream assignment is followed by a blank. In the example above, the three stream assignments are 15, 18, and 18 characters in length so they begin in columns 1, 21, and 41.

When a value is output, it is first converted to a character string and then output in that form; this conversion is discussed in the section on "Value Conversion". In the RANGE program, all of the values have the same data type, namely FLOAT(15). The conversion of such a value to a character string proceeds as follows: the value is converted from BINARY to DECIMAL with an equivalent precision; in this case, the target data type is DEC FLOAT(5). Then the decimal value is converted to a character string in a straightforward way to give a string of 12 characters.

A Second Example of Data-Directed Input/Output

The following program illustrates the remarkable flexibility of data-directed input:

```
UPD:  PROC;
         DCL (SYSIN,SYSPRINT) FILE;
         DCL K CHAR(10);
         DCL 01 M,
                02 NAME CHAR(30) VAR,
                02 ADDRESS,
                   03 STREET CHAR(30) VAR,
                   03 CSZ,
                      04 CITY CHAR(20) VAR,
                      04 STATE CHAR(2),
                      04 ZIP PIC"99999",
                02 EXPIRY,
                   03 MONTH PIC"99",
                   03 YEAR PIC"99",
                02 ACCOUNT PIC"$$$9.99";
         DCL MEMBERS FILE;
         OPEN FILE(MEMBERS) KEYED UPDATE;
         DO WHILE("1"B);
            GET DATA(K);
            IF K="" THEN DO;
               CLOSE FILE(MEMBERS);
               RETURN;
               END;
            READ FILE(MEMBERS) KEY(K) INTO(M);
            PUT DATA(M);
            GET DATA(M);
            REWRITE FILE(MEMBERS) KEY(K) FROM(M);
            END;
      END;
```

This program uses record input/output and therefore anticipates the reader's progress through the manual; however, the program is easily explained.

The program assumes that a record data set exists that describes the membership of an organization, giving the status of annual dues for each member. Each record is accessed by means of a character string, the key, and has a value that can be stored in the structure M in the program. The main part of the program is a loop. Each time through the loop, the program gets a key from the user, stops if the key is an empty character string, uses the key to read the value of a record from the file into the structure M, prints a copy of the value, gets modifications of the value from the user, and writes the modified value on the record.

The interesting part of the program is the statement 'GET DATA(M);'. This statement allows the user to enter modifications to any components of M. For example, the input stream might be as follows:

```
K= "CRIEM06301";
MONTH= 3   YEAR= 74;
K= "MAREM06733";
ZIP= 02139;
```

Two records are changed. For the member whose key is CRIEM06301, the date of membership expiration is changed to March, 1974. For the member whose key is MAREM06733, the zip code is changed to read 02139.

Principles and Exceptions

The underlying principle of data-directed input is as follows: when a stream assignment is read by a GET DATA assignment, it is treated as if it were an assignment statement that appeared exactly where the GET DATA statement appears. For practical reasons, the following exceptions apply:

- The assignment must not require computation. If the target variable has subscripts, they must be signed or unsigned integers. The value on the right must be a value that could appear in a storage unit just as it is; for example, 4+5 is not allowed, but 4+5I is.

- The variable name must designate a scalar storage unit. The variable name must appear explicitly in the DATA list or else it must designate a component of an aggregate variable whose name appears in the DATA list.

- Stream assignments are not separated by semicolons (except where semicolon is used to terminate input). Instead, a sequence of blanks, tab characters, linemarks, and commas can be used. Blanks, tabs, and linemarks can also be used adjacent to the '=' sign.

The underlying principle of data-directed output is as follows: if a PUT DATA is executed for a given list of variables and produces a certain output stream, then the execution of a GET DATA with the same list of variables on an identical input stream will assign to the variables their original values. It is not suggested that one would actually perform this operation; it is a way to store values in a stream, but far inferior to the facilities of record input/output. However, this principle does provide an understanding of the design of data-directed output. For example, a PUT DATA statement adds a semicolon to the end of its stream output so the stream would be delimited if it were used for data-directed input. Exceptions to this principle of reversibility are:

- If the output file has the PRINT attribute, the quote marks are removed from the value of a character string. This allows character values to be used for identification of output but, at the same time, interferes with their being read back as input character strings.

- For all output files, a BINARY arithmetic value is converted to base DECIMAL. This may result in the loss of some precision, so that when the value is converted back during input it will not be quite the same BINARY arithmetic value.

Guidelines for Data-Directed Input/Output

Data-directed input/output is best suited for temporary applications; either in a program written quickly and used only a few times, or as temporary diagnostic output in a program being tested. The latter application of data-directed input/output is a useful debugging aid. Appropriate PUT statements can be inserted to produce dumps of specific data without regard to the format of the output. If each such statement is marked with a comment indicating its role, such as /*DUMP*/, the statements can be systematically removed when debugging is complete.

Data-directed input/output is the most device-independent form of input/output. PL/I arranges stream assignments in a way that is readable and that is usually neatly aligned in columns. Since each value is paired with an identifying variable, no reasonable arrangement of the data can interfere with the proper identification of the values.

Data-directed input/output is well protected against user errors. If the user places a variable in the input stream that does not appear in the DATA option, the NAME condition occurs; and an array subscript that is out of bounds is similarly treated. There is a fairly good chance that a value will either arrive at its intended variable or will be rejected as invalid.

These advantages are balanced by disadvantages. Data-directed input/output is the least efficient of the modes of input/output since the assignment of input data must be determined entirely at execution time. The preparation of input data requires more keystrokes, since each value must be preceded by its variable name. Output data can become cluttered by the repeated occurrences of just a few variables. For all of these reasons, data-directed input/output should be used only for small-scale, simple transmission of data.

LIST-DIRECTED INPUT/OUTPUT

When list-directed input is used, the variable that is the target of an input value is given in a <u>list</u> that is part of the GET statement; so this list determines where the input values go. For this reason, the term "list-directed" is applied to this input/output facility.

An Example of List-Directed Input/Output

The program to calculate the range of an artillery piece is used here as an example of list-directed input/output. That program was given under "Data-Directed Input/Output", and only the two input/output statements need be changed, as follows:

```
RANGE2:  ...
         ...
         GET LIST(V0, THETA);
         ...
         PUT SKIP LIST(V0, THETA, RANGE);
         ...
         END;
```

List-directed input/output is quite similar to data-directed input/output, and the program above is testimony to this fact. RANGE2 differs from RANGE only in the use of the keyword LIST instead of DATA in the input/output statements. Suppose this program is being used to calculate four trajectories, just as RANGE was. Then the input stream would be:

```
1000   35
1000   40
1000   45
1280   45
0      45
```

This input is the bare minimum and looks more like computer input prepared in bulk, on punched cards, for example. The input is nicely formatted, but PL/I is not influenced by that. If, by mistake, the stream pointer is just <u>after</u> the 1000 on the first line, the whole run of the program will be invalid; indeed, it will not stop until a zero is encountered somewhere beyond the portion of the stream shown above or until some input/output condition stops it.

It is possible to ignore a target in the GET statement by leaving a value blank. In order to do so, however, commas must be used to separate the values, as follows:

```
1000,   35,
    ,   40,
    ,   45,
1280,     ,
   0,     ,
```

Again, the format is for the benefit of the user, not PL/I. The stream could be entered as

```
1000,35,,40,,45,1280,,0,,
```

with exactly the same effect. The use of a comma as a separator is optional in list-directed input, but it is suggested that commas be used throughout when one or more targets are ignored.

If the input stream is the one just discussed, the program produces the following addition to the output stream:

```
1.0000E+003        3.5000E+001        2.9207E+004
1.0000E+003        4.0000E+001        3.0609E+004
1.0000E+003        4.5000E+001        3.1081E+004
1.2800E+003        4.5000E+001        5.0923E+004
```

This output will be labelled if the following statement is inserted before the loop:

```
PUT SKIP LIST("ßVELOCITYßßß", "ßELEVATIONßß", "ßRANGEßßßßßß");
```

Note that each heading is padded with blanks so it is just as wide as the value it labels; this assures that it will line up whatever tab stops are used. The blank at the beginning of each heading corresponds to the sign position of the numbers. The statement prints the line:

```
VELOCITY          ELEVATION          RANGE
```

at the beginning of the output.

Compound List Items

In its simplest case, a GET LIST statement has a list of designators of scalar values, and these are paired, one-for-one, with the values in the input stream. The GET LIST statement in the RANGE2 program was of this sort: its list was composed of two scalar names, V0 and THETA. However, list-directed input is not restricted to this simple case; instead, an item can be any variable name, scalar or aggregate, or it can be a parenthesized iterated list of items. Such items are interpreted by expanding them from a single item into a sequence of items; and they are therefore called compound items.

An item that is an array variable name represents the elements of the array listed in row-major order. Suppose the following declaration is in effect:

```
DCL A(3:4,3) FLOAT;
```

Then the statement

```
GET LIST(A);
```

is interpreted as

```
GET LIST(A(3,1), A(3,2), A(3,3), A(4,1), A(4,2), A(4,3));
```

and therefore reads six values from the input stream.

An item that is a structure variable name represents the members of the structure listed in the order in which they are declared. If a member of the structure is not a scalar, then the member is expanded, in turn, into a list of its components. Suppose the following declaration is in effect:

```
DCL 01 SPECS,
       02 SIDE(2),
          03 H FLOAT,
          03 W FLOAT,
       02 AREA FLOAT;
```

Then the statement

```
GET LIST(SPECS);
```

is interpreted as

```
GET LIST(SPECS.SIDE(1).H, SPECS.SIDE(1).W,
         SPECS.SIDE(2).H, SPECS.SIDE(2).W, SPECS.AREA);
```

and therefore reads five values from the input stream. The item in this example is a "level one" structure; but any component of an aggregate can be referred to. For example, GET LIST(SPECS.SIDE) is interpreted as above but with SPECS.AREA omitted.

An item that is a parenthesized iterated list represents a list of subscripted items. For example, the statement

```
GET LIST((X(I), Y(I) DO I = 2 to 4));
```

is interpreted as

```
GET LIST(X(2), Y(2), X(3), Y(3), X(4), Y(4));
```

and therefore reads six values from the input stream. The form of the iterated list is based on the DO statement, and can use any of the multiple do clauses defined in the section on "Program Flow". The following example illustrates the options allowed:

```
GET LIST((BUSY(K) DO K = 1, 5, 6 REPEAT 2*K WHILE(K<25),
                         23 TO 15 BY -2));
```

This statement uses the following sequence of subscripts:

```
1, 5, 6, 12, 24, 23, 21, 19, 17, 15
```

Therefore it reads ten values from the input stream into the array BUSY.

All of the compound items have now been informally described; but without further examples, certain useful techniques might be overlooked by the reader. For example, an item in a parenthesized iterated list can itself be a compound item. The statement

```
GET LIST((Q(I,*) DO I = 1 TO 3));
```

reads in values for the first three rows of the array Q. In this example, Q(I,*) is a compound item representing the values in a row of the array Q.

An item in a parenthesized iterated list can be another parenthesized iterated list, as shown in the following statement:

```
GET LIST(X, ((A(J,K) DO K = 1 TO 2), B(J) DO J = 3 TO 4), Y);
```

This statement is interpreted as

```
GET LIST(X, A(3,1), A(3,2), B(3), A(4,1), A(4,2), B(4), Y);
```

and therefore reads eight values from the input stream.

Any appropriate expressions can be used in the clauses of the multiple do. For example, the statement

```
GET LIST(X, (Y(M) DO M = M1 TO RAD*U(J-1)));
```

can be used, but of course its interpretation cannot be determined until, as the result of the execution of earlier statements, values have been assigned to M1, RAD, J, and U(J-1). Even values previously input by the same GET statement can be used to control the statement; for example, the statement

```
GET LIST(N, (W(I) DO I = 1 TO N));
```

reads a value into N from the input stream and then uses that value to determine how many values are read into the array W.

The list of a PUT LIST statement can use the compound items that have been described for the GET LIST statement. Each compound item in a PUT LIST statement is interpreted as a sequence of items in exactly the way it would be interpreted in a GET LIST statement.

Each output value in a PUT LIST statement can be given by an expression of unlimited complexity. Just as a GET LIST allows the use of anything that could appear on the left side of an assignment statement, so a PUT statement allows use of anything that could appear on the right side of an assignment statement.

Pseudo-Variable List Items

Since an item in a GET LIST statement can be any target, it can be a pseudo-variable. For example, the statement

```
GET LIST(REAL(X),IMAG(X));
```

reads two values from the input stream and assigns them as the real and imaginary parts of X, respectively. The values must be REAL and the variable X must be COMPLEX.

Principles and Exceptions

The principle of list-directed input is as follows: the variable names in the list in the GET statement are processed from left to right; and, for each variable, an assignment statement is executed that consists of the variable name, an '=' sign, and the next item from the input stream. In practice, list-directed input is both more powerful and less powerful than an ordinary assignment statement, as indicated by the following exceptions:

- When a variable designates an aggregate, a sufficient number of (scalar) values are taken from the input stream to provide a complete value for the variable, as described under "Compound Items", above. There is no corresponding feature of an ordinary assignment statement because there is no way to write an aggregate constant. For example, one cannot write 'A = 89,-99,17;' to assign an array value to A.

- Items read from the data stream must be values as they stand, without further computation.

- The reference that appears in a GET LIST list must have an arithmetic or string data type since only values of these types can appear in the input stream.

The principle of list-directed output is the same as that for data-directed output; whatever is output by a given statement can be input by a statement with the same list. As with data-directed input/output, the exceptions to this principle occur when character strings are output under the PRINT attribute and when a BINARY value is converted to DECIMAL for output.

Guidelines for List-Directed Input/Output

List-directed input/output is more efficient than either data-directed or edit-directed input/output because it does not require the execution-time interpretation of variable names (as does data-directed input/output) or execution-time editing (as does edit-directed input/output). On grounds of efficiency alone, then, it is preferred; but its limited and rigid format is a disadvantage, especially for output.

For list-directed output, the programmer cannot choose a format freely; he must take into account the tab stops provided on the printer, for example. Further, automatic conversion of arithmetic values to character strings uses moderately complicated rules and can produce some surprises. Thus, it is often easier to use edit-directed output from the outset and be assured of full control over the output format.

The use of list-directed input is more attractive, since the insensitivity to format is an advantage. If a large volume of input is required, it can be typed or punched value after value, line after line, using exactly the precision and scaling that appear in the raw data.

EDIT-DIRECTED INPUT/OUTPUT

When edit-directed input/output is used, the statement includes instructions for the editing of the transmitted values. The edit-directed statement includes a list of data items, just as a list-directed statement does, but also has a list of format items that control the editing of each value, either checking the format of input or supplying the format for output. For this reason, the term "edit-directed" is applied to this input/output facility.

Examples of Edit-Directed Input/Output

Edit-directed input/output will be introduced through discussion of several versions of a single program. The program inputs a number, divides it by two, and outputs both the given number and the computed result. It is known that the input is supplied as a signed, three-digit number. The first version of the program is:

```
P1:  PROC;
     DCL (SYSIN,SYSPRINT) FILE;
     DCL (X,Y) FLOAT;
     GET EDIT(X)(A(4));
     Y = X/2;
     PUT EDIT(X,Y)(SKIP,A,X(3),A);
     END;
```

This program uses the most fundamental of all format items, the A (for "alphanumeric") format item. This format item specifies the reading of characters from the input stream without any checking or conversion whatever.

The GET statement is interpreted as "read the next four characters from the input stream and assign them (as a character-string value) to X". It performs a very simple input operation. A conversion of data type does occur, but it is part of the assignment of the character string input to X, and it is not controlled by the GET statement. Suppose the input stream currently begins with '-709'; then '-709' is expressed as a REAL BINARY FLOAT(27) value and assigned to X. The computation part of the program assigns -354.5 to Y.

The PUT statement is interpreted as "skip to the beginning of a new line; output the value of X as a character string; output three blanks; and output Y as a character string". Since the values of X and Y are not character strings, these values are converted to character strings before output. The output is the line

```
-7.09000000E+002␣␣␣-3.54500000E+002
```

The first character of this line is in column 1 of the output medium.

The PUT statement in this example requires further explanation. While the interpretation given for the statement is accurate, it may not be obvious how that interpretation was obtained. The two lists in an edit-directed statement are processed in parallel. That is, the sequence of processing is determined by the data list, but a reference is made to the format list for each item in the data list. When the next item in the format list is a data format item, such as A, it tells how the data value is to be processed, and the reference is complete. However, when the next item in the format list is a control format item, such as SKIP or X(3), it does not tell how to process the data item; and the control format item is executed and a new reference is made to the format list.

The program just given exercises a minimum of control over its input; it merely requires four characters that can be converted into an arithmetic value. Even though it was stated earlier that the input would be supplied as a signed, three-digit integer, this program accepts '␣␣+6' or '.003' or even the very large number '7E30'. The program shows a similar indifference to the format of the output, and leaves this format entirely to the built-in rules for conversion of an arithmetic value to a character-string value.

The following version of the program uses pictured character-string variables to exercise the proper control of input and output:

```
P2:  PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL IN PICTURE"S999";
        DCL OUT PICTURE"-999.V9";
        GET EDIT(IN)(A(4));
        OUT = IN/2;
        PUT EDIT(IN,OUT)(SKIP,A,X(3),A);
        END;
```

The variable name IN is declared PICTURE"S999", and is thus restricted to precisely the signed, three-digit integer supplied as input when the program is properly used. Similarly, the variable name OUT is declared PICTURE"-999.V9" to provide a good format for the computed result. If the string '-7.9' is supplied, the CONVERSION condition occurs because this value cannot be assigned to IN; and thus the error is detected. When the input is '-709', the program runs to completion and outputs the line

```
-709␣␣␣-354.5
```

which is much more readable than the output from the first version.

The program just discussed controls input/output well, but uses pictured character-string values for its computation instead of the floating-point variables used in the first version. This mode of computation causes a considerable loss of efficiency, especially if the computation is not so trivial. A third program combines well-controlled input/output with efficient computation:

```
P3:  PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL IN PICTURE"S999";
        DCL OUT PICTURE"-999.V9";
        DCL (X,Y) FLOAT;
        GET EDIT(IN)(A(4));
        X = IN;
        Y = X/2;
        OUT = Y;
        PUT EDIT(IN,OUT)(SKIP,A,X(3),A);
        END;
```

The version of the program just given uses the pictured variable names IN and OUT in a rather specialized way; IN is an intermediary between the input stream and the computation-oriented variable X, and OUT is an intermediary between Y and the output stream. PL/I has a special facility, the picture format item, for this situation, as the following, final, version of the program shows:

```
P4:  PROC;
        DCL (SYSIN,SYSPRINT) FILE;
        DCL (X,Y) FLOAT;
        GET EDIT(X)(P"S999");
        Y = -X/2;
        PUT EDIT(X,Y)(SKIP,P"S999",X(3),P"-999.V9");
        END;
```

This program means exactly the same thing as P3 did, and therefore needs no interpretation. Observe that it is identical to P1 except for the use of the picture format items instead of the A format items.

The preceding examples have ranged from the simplest of the data format items, the character-string format item, to the most powerful, the picture format item. The examples have also shown two representatives, SKIP and X, of the control format items. Finally, the examples have shown how the data list refers to the format list for the specification of format. The remainder of this discussion covers this ground again, supplying a complete description of the PL/I edit-directed input/output facility.

Data Format Items

In an edit-directed statement, each data item must have a corresponding data format item. Each data format item describes a field; that is, a sequence of characters either read from the input stream or added to the end of the output stream. Usually, the format item gives the width of the field; that is, the number of characters, including blanks, contained in the field. In addition, the format describes the way the value is represented in the field; that is, it gives the format of the value.

The format imposed on the input stream may or may not be very precise. For example, the format item P"S999" means, quite precisely, that "the next four characters of the input stream must be a signed integer with three digits". On the other hand, F(10) means, less precisely, that "the next ten characters of the input stream must contain a fixed-point value representation, signed or unsigned, with or without a decimal point, filling the whole field or sharing it with leading or trailing blanks".

For output, the role of the format item is to supply the format of the output value representation. Again, the format may or may not be very precise. For example, E(20,8,9) means "output four blanks and a floating-point value representation that has a signed, nine-digit mantissa with eight digits to the right of the decimal point". On the other hand, the format item A means "output however many characters are necessary to represent the given value".

The occurrence of linemarks and pagemarks in the stream are ignored during the processing of a data format item. During input, the field length may exceed the number of character positions that remain on the current line of the input stream; in this case, additional character positions in the subsequent lines are used. During output, the field length may require character positions beyond the number allowed by "linesize". In this case, new lines are created to supply additional character positions. If the new lines would exceed the number of lines allowed on the page by "pagesize", the "endpage" condition occurs and (when the condition has been processed) a new line is begun. Thus if the programmer wishes to ignore all or part of the layout of the output stream, he can do so and PL/I will process the layout automatically.

There is a format item for each of the main types of computational data. They are as follows:

Name	Format Item
character-string	A(w)
bit-string	B(w)
fixed-point	F(w,fw,dm)
floating-point	E(w,fw,ms)
complex	C(part,part)
picture	P"x"

In these format items, w (the width), fw (the fraction width), dm (the decimal multiplier), and ms (the mantissa significance) can each be any expression whose value can be converted to an integer. Except for the decimal multiplier, dm, the integer value must be positive or zero. The x in the last format item represents a picture, as described earlier, in the section on "Value Storage". Often it is not necessary to give all the arguments, as the following examples will show.

STRING FORMAT ITEMS

The stream representation of a string value is processed by the string format items. For character strings, the allowed forms are

```
A(w)      -- used for both input and output
A         -- used for output only
```

For bit strings, the allowed forms are

```
B(w)      -- used for both input and output
B         -- used for output only
```

The processing of the character-string and bit-string format items are parallel in every respect.

String Input

Examples of these format items for input follow:

Input Stream	Format Item	Inbound Value	Comment
2.5ß	A(4)	"2.5ß"	The next four characters are read,
ßßßß	A(4)	"ßßßß"	whatever they are.
ß+3.0-2.3I	A(10)	"ß+3.0-2.3I"	The next ten characters are read.
010	B(3)	"010"B	Only 0 and 1 are accepted.
ß1ß	B(3)	"1"B	The blanks are deleted.
ßßß	B(3)	""B	The string can be empty.
000	B(3)	"000"B	Three zeroes are three bits.
012	B(3)	(CONV)	Only bits, leading blanks, and
"01"B	B(5)	(CONV)	trailing blanks can appear.

String Output

During output, a string value is left adjusted; that is, if the string value does not fill the field, it is placed as far to the left as possible and the remainder of the field is filled with blanks. This contrasts with the output of arithmetic values, which are right adjusted. When w is omitted from a string format item, the width of the output field is determined by the length of the output string value. This is the only case in which the field width is not given explicitly in a format item. Examples of the use of these format items for output follow:

Outbound Value	Format Item	Output Stream	Comment
"ABC"	A(5)	ABCßß	Blanks are supplied at the right.
"ßßABC"	A(5)	ßßABC	Given blanks remain in the string.
""	A(5)	ßßßßß	A null string is accepted.
"ABC"	A	ABC	The value determines the width.
"1"B	B(5)	1ßßßß	Blanks are supplied at the right.
""B	B(5)	ßßßßß	No bits in a null bit string.
"00B	B	00	The value determines the width.
"ßßABCßß"	A(5)	(STRINGSIZE)	Seven characters do not fit in the field.
"012"	B(3)	(CONV)	Only bits are allowed for B(3).

FIXED-POINT FORMAT ITEMS

The stream representation of a fixed-point value is processed by a fixed-point format item, which can have any of the following forms:

F(w) -- used for input or output
F(w,fw) -- used primarily for output
F(w,fw,dm) -- used for special applications

The value of w (width) determines the size of the field. The value of fw (fraction width) determines the number of fractional digits in the representation. The value of dm (decimal multiplier) determines a multiplier for the transmitted value.

Fixed-Point Input

Usually the form F(w) is used for input and the fraction-width and decimal-multiplier arguments are omitted. Input is performed as follows:

- A character string of w characters is read from the input stream. The string must contain an optionally-signed, real, fixed-point constant or else must be entirely blank.

- The input string is assigned to an intermediate variable of data type REAL DECIMAL FIXED(p,q). If (1) the input constant is DECIMAL, (2) the input constant has a precision and scale-factor within the maximums of GCOS PL/I, and (3) fw and dm are omitted from the format item, then the precision attribute (p,q) is taken directly from the input constant.

- The value of the intermediate variable is assigned to the target given in the data list.

In the steps just given, two assignments are made; one from the stream to an intermediate variable and a second from the intermediate variable to the target variable. These assignments are performed as if they arose from an assignment statement; that is, the necessary conversion is performed and, if the assignment or the conversion cannot be performed for some reason, the appropriate condition occurs. The intermediate variable exists only long enough to convey the value to the target, and is not used in any other way.

Examples of the processing of a five-character input field by a fixed-point format item follow.

Input Stream	Format Item	Intermediate Value	Comment
-7.2␣	F(5)	-7.2	The position of the value
␣7.2␣	F(5)	+7.2	representation in the field
␣␣7.2	F(5)	+7.2	does not affect its
7.2␣␣	F(5)	+7.2	interpretation.
␣␣␣␣␣	F(5)	+0.	If the field is blank, its value is 0.
	F(0)	+0.	If w=0 there is no input, but value is 0.
72E-1	F(5)	(CONV)	If the input is not a valid fixed-point
-7␣.2	F(5)	(CONV)	value representation, the CONVERSION
7.2DB	F(5)	(CONV)	condition is signalled.
72␣␣␣	F(5,2)	+.72	fw=2 gives two fractional digits, and
72␣␣␣	F(5)	+72.	fw=0 (default) gives none.
7.2␣␣	F(5,2)	+7.2	But when a decimal point is in the
7.2␣␣	F(5)	+7.2	stream, fw is ignored.
7.2␣␣	F(5,2,-3)	+.0072	dm=-3 multiplies input by 10**-3 = .001
7.2␣␣	F(5,2,3)	+7200.	dm=3 multiplies input by 10**3 = 1000
72␣␣␣	F(5,2,3)	+720.	fw=2 gives two fractional digits, then dm=3 multiplies by 1000.

The last two groups of examples show the use of a nonzero fraction width or decimal multiplier. Such format items should be used only when there is a clear justification for accepting input values that are not "true" values; this might occur, for example, when input is being prepared by an automatic device that can only produce a sequence of digits (but no decimal point or scale factor) on its output medium.

When fw is omitted, it is assumed to be zero; therefore, a value representation without a decimal point is treated as an integer, which is the everyday convention. When dm is omitted, it is also assumed to be zero; therefore, the value is multiplied by 10**0 = 1. Thus the defaults are chosen so they leave the transmitted value unchanged.

Fixed-Point Output

Usually the form F(w) or F(w,fw) is used for output, and the multiplier is omitted. When the form F(w) is used, it is assumed that fw = 0. An intermediate variable is used with output in the same way as was previously described for input. The data type of the intermediate variable is REAL DECIMAL FIXED(p,q). The scale factor, q, is fw and the significance, p, is as large as w allows. That is, p is obtained by reducing w by one if necessary to allow for a minus sign and (when fw is not zero) by one more to allow for a decimal point; however, p cannot exceed the maximum GCOS PL/I precision, 59. The outbound value is converted to an attractive, right-adjusted value representation in the output stream, as shown in the following examples:

Outbound Value	Format Item	Output Stream	Comment
-3.	F(5,2)	-3.00	fw=2 provides two fractional digits, and
-3.	F(5)	ßßß-3	fw=0 (default) provides none.
-3.	F(5,3)	(SIZE)	-3.000 does not fit in the field.
+3.	F(5,3)	3.000	3.000 (without sign) does fit.
+17.46	F(5,2)	17.46	fw=2 fits the exact value (in this case),
+17.46	F(5,1)	ß17.5	fw=1 rounds to one fractional digit, and
+17.46	F(5)	ßßß17	fw=0 rounds to an integer.
+17.46	F(5,-1)	(ERROR)	(fw must be nonnegative)
+0.	F(5,2)	ß0.00	There are many ways
+0.	F(5)	ßßßß0	to print zero.
+17.46	F(5,3,-1)	1.746	dm=-1 multiplies value by 10**-1 = .1
+17.46	F(5,0,2)	ß1746	dm=2 multiplies value by 10**2 = 100
+17.46	F(5,0,-1)	ßßßß2	then fw = 0 causes rounding.

The last group shows the use of a nonzero decimal multiplier with output. The use of this feature should be restricted to the applications that are similar to its use with input; that is, it should be used to change the "true" stored value of a number to some scaled output form for a specialized application.

FLOATING-POINT FORMAT ITEMS

The stream representation of a floating-point value is processed by the floating-point format item, which can have any of the following forms:

```
E(w)          -- used for input or output
E(w,fw)       -- used primarily for output
E(w,fw,ms)    -- used only for output
```

The value of w (the width) determines the size of the field. The value of fw (fraction width) determines the number of fractional digits in the mantissa of the representation. The value of ms (mantissa significance) determines the number of digits in the entire mantissa.

Floating-Point Input

Usually the form E(w) is used for input and the fraction-width and mantissa-significance arguments are omitted. Input is performed as follows:

- A character string of w characters is read from the input stream. The string must contain (1) an optionally-signed, real constant that is either fixed-point or floating-point or (2) a sequence of blanks, which is interpreted as zero.
- The string is assigned to an intermediate variable of data type REAL DECIMAL FLOAT(p), where p is the precision of the input constant.
- The value of the intermediate variable is assigned to the corresponding target in the data list.

The role of the intermediate variable used here is the same as that used with the fixed-point format item. It exists only to convey the input to the target with the required conversions.

Examples of the processing of a seven-character input field by a floating-point format item follow:

Input Stream	Format Item	Inbound Value	Comment
1.3E7ßß	E(7)	+1.3E+7	A floating-point or fixed-point value
ß-50+4ß	E(7)	-50.E+4	representation is accepted at any
ßßßß5ßß	.0E(7)	+5.0E+0	position in the input field.
ßßßßßßß	E(7)	+0.0E+0	If the field is blank, its value is 0.
	E(0)	+0.0E+0	If w = 0, no input occurs, but value is 0.
1.3E7.0	E(7)	(CONV)	If the input is not valid, the
-50+ßßß	E(7)	(CONV)	CONVERSION condition occurs.
ßßß13E7	E(7,2)	+.13E7	fw = 2 gives two fractional digits, and
ßßß13E7	E(7)	+13.E7	fw = 0 (default) gives none.
ßß1.3E7	E(7,2)	+1.3E7	But when a decimal point is in the
ßß1.3E7	E(7)	+1.3E7	mantissa, fw is ignored

The last four examples show the use of a nonzero fraction width, fw. When an input constant does not contain a decimal point, fw supplies the position for an assumed decimal point. As in the case of the fixed-point format item, such floating-point format items should be used only when there is a clear justification for accepting input values that are not "true" values.

Floating-Point Output

For output, the data type of the intermediate variable is REAL DECIMAL FLOAT(p), where p is the precision derived from the data type of the value supplied by the data item. All three forms of the E format are commonly used. Omitted arguments are interpreted as follows:

E(w,fw)	means	E(w,fw,fw+1)
E(w)	means	E(w,p-1,p)

Thus, when the mantissa significance is not specified in the format item, it is calculated so that the mantissa has a one-digit integer part. When neither fw nor ms is given, the precision of the output value itself, p, is used. The outbound value is converted to an attractive representation in the output stream, as shown in the following examples:

Outbound Value	Format Item	Output Stream	Comment
7.5	E(11,2,4)	␣75.00E-001	When both fw and ms are given, the
7.5	E(11,0,4)	␣␣7500E-003	decimal can be adjusted.
7.5	E(11,3,3)	␣0.750E+001	Note leading zero when fw=ms.
7.5	E(11,4,4)	0.7500E+001	Fits because there is no sign
7.5	E(11,3)	␣7.500E+000	When ms is not given, the
-7.5	E(11,3)	-7.500E+000	mantissa has fw+1 digits, so there
750	E(11,3)	␣7.500E+002	is one integer digit
0	E(11,3)	␣0.000E+000	Zero has a zero exponent.
+7.5E+0	E(11)	␣␣␣7.5E+000	When fw and ms are not given, the
+7.500E+0	E(11)	␣7.500E+000	precision is taken from the value itself.

COMPLEX FORMAT ITEMS

The stream representation of a complex value is processed by the complex format item, which can have either of the following forms:

```
C(part)          -- used for input or output
C(part1,part2)   -- used for input or output
```

The part, part1, or part2 can be any format item for a real value; that is, one of the following:

```
fixed-point format item
floating-point format item
picture format item
```

This format item always describes two values in the stream: the real and imaginary parts of the complex value. If only one part is given, it is used twice. Although imaginary values are usually followed by I in PL/I, this is not the case when the complex format item is used.

Complex Input

Examples of the use of this format item follow:

Input Stream	Format Item	Inbound Value	Comment
␣␣3␣-2.3	C(F(3),F(5))	+3.-2.3I	The field is treated as
␣3␣␣␣-2.3␣	C(F(5))	+3.-2.3I	two fields.
+2.32E-3-6.80E-3	C(E(8))	+2.32E-3-6.80E-3I	
␣3␣␣␣␣␣␣␣␣	C(F(5))	+3.+0.I	A blank part is a 0.
␣3␣␣␣-2.3I	C(F(5))	(CONV)	I is not allowed.

Complex Output

Examples of the use of the complex format item to process output follow:

Outbound Value	Format Item	Output Stream
+3.0+2.3I	C(F(5,2))	␣3.00␣2.30
+2.32E-3-6.80E-3I	C(E(11,2),E(11,2,2))	␣␣2.32E-003␣-0.68E-002
+0.0E+0-3.0E+2I	C(F(8,2))	␣␣␣␣0.00␣-300.00
+3.0-2.3I	C(F(4,2))	(SIZE)
+3.0+2.3I	C(F(4,2))	3.002.30

PICTURE FORMAT ITEMS

The stream representation of any computational value (except COMPLEX) can be processed by an appropriate picture format item, which has the form

P"x"

where x is any of the pictures described under "Pictured String Storage" in the section on "Value Storage".

When input is performed under control of the picture format item, the following steps are performed:

- The length of the character string described by the picture is determined, and a string of that length is read from the input stream and assigned to an intermediate variable of data type PICTURE"x".
- The value of the intermediate variable is assigned to the target variable.

Both of the assignments performed during the picture-format input require further comment. The first assignment changes the data type of the input string, but never changes the string itself. Suppose the input stream supplies the characters -709 and the format item is P"S999"; then the input value is "-709" and has the data type CHARACTER. After assignment to the intermediate variable, the value is still -709, but it now has the data type PICTURE"S999". Thus the string has been checked for conformity with the picture and has been given the interpretation associated with the picture.

The second assignment entails the conversion of the value of the intermediate pictured variable to the data type of the target value. Since the data type of the target can, in various cases, be any PL/I data type, there are many possible conversions.

When output is performed under control of a picture format item, the following steps are performed:

- The value supplied by the source expression in the data list is assigned to an intermediate variable of data type PICTURE"x", where x is the picture given in the picture format item.
- The value of the intermediate variable, which is already a character string, is added to the output stream without being changed.

Just as with input, two assignments are made; one from the source expression in the data list to the intermediate variable, and a second from the intermediate variable to the output stream. If an assignment cannot be performed, the appropriate PL/I condition occurs.

The pictures are classified into three groups: fixed-point, floating-point, and character. A fixed-point picture describes a character string that has the form of an optionally-signed, fixed-point constant. A floating-point picture describes a character string that has the form of an optionally-signed floating-point constant. A character picture describes a character string that might not be suitable for interpretation as an arithmetic value but could be useful as an identifier of some kind. Examples of picture format items from each classification are now given. The commentary given with the examples is an informal summary of the definitions given for pictures in the section on "Value Storage".

Fixed-Point Pictures

The fixed-point picture format item is considered here, and many examples are given. For each 9 in a picture, a digit appears in the corresponding character position of the stream; for an S, a sign ('+' or '-' but not a blank); and for a '-', a blank or a '-' (but not a '+').

Stream	Format	Internal	Comment
-823	P"S999"	-823.	A signed, three-digit integer with various signs.
+823	P"S999"	+823.	
⌀823	P"-999"	+823.	
-823	P"-999"	-823.	

The examples above should be read in both directions. For example, the first line should be read first as the transmission of the characters '-823' from the input stream to a target, and then read a second time as the transmission of the value of an internal source with value '-823.' to the output stream. This approach will be used for subsequent examples of the picture format item.

A Z matches a digit; but if the digit would be a leading zero, it is suppressed (replaced by blank) on output and may or may not be suppressed (at the option of the user who prepares the data) on input. A sequence such as SSS is like SZZ except that the sign "drifts" to the right when leading zeroes are suppressed. The sequence '---' represents a "drifting" minus sign in the same way.

Stream	Format	Internal	Comment
⌀⌀68	ZZZ9	+0068.	Zero suppression and drifting signs for input or output
⌀⌀⌀5	ZZZ9	+0005.	
⌀⌀⌀⌀	ZZZZ	+0000.	
⌀+68	SSS9	+0068.	
⌀⌀-5	---9	-0005.	

Observe that a blank field can appear in the stream, but only when there is no 9 in the picture.

In ordinary applications of the picture format item, the decimal point is indicated in the picture by two characters, V., which matches the '.' in the stream.

Stream	Format	Internal	Comment
-53.60	SSSV.99	-053.60	Ordinary decimal
⌀⌀-.60	SSSV.99	-000.60	points for input
⌀⌀829.	----9V.	+00829.	or output.

When V and '.' are not adjacent in the picture, the transmitted value is changed. The V indicates the position of the decimal point in the internal representation of the value, and the '.' indicates the position of the decimal point in the stream representation. Details are given in the section on "Value Storage".

A parenthesized integer can be used in a picture to indicate repetition of the following picture character. For example,

P"S(7)9"	means	P"S9999999"
P"(5)SV.(2)9"	means	P"SSSSSV.99"

The parenthesized integer must be a constant; that is, it cannot be written as an expression and computed when the program is executed.

When one of the assignments in the interpretation of a picture format item would lose a digit at the left end of the value, the SIZE condition occurs. But when an assignment would lose a digit at the <u>right</u> end of the value, that digit is truncated, without warning, and no condition occurs. Suppose the target of input or the source of output has the data type DEC(6,2). The following examples show various instances of digit-loss:

Stream	Format	DEC(6,2)	Comment
-9.2362	P"-9.9999"	(SIZE)	Input error.
-9.2362	P"-9V.999"	-0009.23	Input approximation.
(SIZE)	P"S999"	+8264.00	Output error.
+82	P"S99"	+0082.99	Output approximation.

The most remarkable aspect of the picture is its handling of commercial symbols. A '$' can be used and can "drift" to the right as a sign can do. Commas can be used and a comma is suppressed when an adjacent leading zero is suppressed. The suffixes CR and DB are allowed.

Stream	Format	Internal	Comment
$129.88	$999V.99	+129.88	Commercial symbols for input and output.
␣␣$6.50	$$$9V.99	+006.50	
$2,619	$$,$$$	+2619.	
␣␣␣$81	$$,$$9	+0081.	
$9.28CR	$9V.99CR	-9.28	

These examples of the fixed-point picture format items do not exhaust all the possibilities, but the omitted possibilities are less frequently used. For a complete description of fixed-point pictures, see the section on "Value Storage".

Floating-Point Pictures

The mantissa of the floating-point picture can be any fixed-point picture that does not contain the commercial symbols, '$', CR, and DB. The exponent picture can have S or '-' as its sign or the sign can be omitted; and up to three digits can be used, with Z for leading digits if desired. If an E is not wanted between the mantissa and the exponent, K is used in the picture instead of E and nothing appears in the stream.

Stream	Format	Value
+3.939E+002	P"S9V.999ES999"	+393.9
␣3.939E+02	P"-9V.999ES99"	+393.9
␣3.939+02	P"-9V.999KS99"	+393.9
␣3939-02	P"-9999KS99"	+39.39

Character Pictures

A character picture can be any sequence of the characters X (matches any character), A (matches any letter, upper or lower case, or blank), or 9 (matches any digit or blank). The picture must not be all nines, since it would be a fixed-point picture in that case.

Stream	Format	Value	Comment
3/MAY/74	P"9XAAAX99"	"3/MAY/74"	Character string, input and output
29AXQ6	P"99AAAX"	"29AXO6"	
␣␣␣␣␣␣	P"99AAAX"	"␣␣␣␣␣␣"	

A 9 can match a blank only in a character picture.

Control Format Items

In an edit-directed statement, provision must be made for those positions of the stream that appear between the value representations. Specifically, the contribution to layout made by the blanks, linemarks, and pagemarks must be taken into account. The control format items are provided for this purpose. Two examples of control format items were given in the example program, namely SKIP (start a new line) and X(3) (skip three character positions).

An output stream with the attribute PRINT can be viewed as divided into pages and lines and character positions. Any other stream, whether for input or output, is divided only into lines and character positions within the lines. A character string that is used as a substitute for a stream (by means of the STRING option of a stream input/output statement) is a single line that is divided only into character positions.

When an input stream is open, it has a stream pointer associated with it. The stream pointer indicates the next character position that will be read (or skipped) by the next input operation. In contrast, an output stream is created as output is performed; that is, character positions as well as the characters themselves are added to the end of the stream. But it is legitimate and very useful to speak as if the output were created in advance as a sequence of blank character positions arranged in lines and pages. This convention allows the use of a stream pointer with an output stream and permits language such as "advance the stream pointer to the first character position in the third line after the current line".

There are five control format items, as follows:

X(e)	--	skip e character positions
COLUMN(e)	--	skip to column e of a line
SKIP(e)	--	skip e lines
LINE(e)	--	skip to line e of a page
PAGE	--	skip to the next page

The e in each of these format items can be any expression whose value can be converted to an integer. In all cases, the value of e must be positive or (for X and SKIP only) zero.

THE X FORMAT ITEM

The X format item has the form

X(e)

Let n be the value of the expression e for a given execution; then the item moves the stream pointer forward by n character positions, proceeding from line to line or page to page if necessary. If n=0, then the item does nothing.

THE COLUMN FORMAT ITEM

The COLUMN format item has the forms

COLUMN(e)
COL(e)

Let n be the value of the expression e for a given execution; then the item advances the stream pointer to the next character position that is in column n; that is, to a character position that is the nth character position of a line. This interpretation implies that if the stream pointer is beyond the nth column the operation is applied to the next line. If n exceeds the length of the line (so the specified character position does not exist), the stream pointer is set to the beginning of the next line.

THE SKIP FORMAT ITEM

The SKIP format item has the form

SKIP(e)

Let n be the value of the expression e for a given execution; then the item moves the stream pointer to the first character position of the nth line after the current line. If n = 0, then the stream pointer is set back to the beginning of the current line and the stream is prepared for overprinting of the current line; but this case is allowed only for an OUTPUT PRINT stream.

THE LINE FORMAT ITEM

The LINE format item has the form

LINE(e)

Let n be the value of the expression e for a given execution; then the item moves the stream pointer to the next character position that is the first character position of the nth line of a page. If the stream pointer is already at such a character position, the stream pointer is not moved. If the stream pointer is at a line whose number is greater than n, the stream pointer is moved to the first line of the next page.

THE PAGE FORMAT ITEM

The PAGE format item has the form

PAGE

The item moves the stream pointer forward to the first character position of the next page.

A control format item can be used only where its use would be reasonable. Any control format item can be applied to an output stream with the PRINT attribute because it has lines and pages. The LINE and PAGE items cannot be applied to a stream that is not a PRINT output stream because such a stream is not divided into pages. The SKIP, LINE, and PAGE items cannot be applied when the STRING option is used because a pseudo-stream is not divided into pages or lines.

A control format item is executed only when PL/I is "on the way" to a data format item; that is, when PL/I is prepared to output a value and is reading through the format list toward the next format data item.

Format Lists

In its simplest form, a format list is a sequence of format items separated by commas. However, there are three facilities for enhancing the form of a format list; namely, the remote format item, the iterated format item, and the "end-around" repetition. These facilities are discussed in the following paragraphs.

REMOTE FORMAT ITEMS

The remote format item has the form

R(ref) -- use a remote format list

The ref must be a reference that has a scalar format value; that is, a value that designates a FORMAT statement. Let fx be the format list in the designated FORMAT statement. When R(ref) is executed, the scanning of the format list in which the remote format item appears is suspended and format items are taken from fx until the end of fx is reached.

ITERATED FORMAT LISTS

The iterated format list can have any of the following forms:

```
int  item    -- constant iteration of an item
int  (fl)    -- constant iteration of a format list
(e)  item    -- computed iteration of an item
(e)  (fl)    -- computed iteration of a format list
```

In these forms, int is an unsigned integer, item is a data, control, or remote format item, fl is a format list, and e is any expression whose value can be converted to an integer. Suppose the value of int or e (whichever is present) is n. Then the iterated format list is interpreted as a sequence of format items composed of n repetitions of item or fl (whichever is present). If n is zero, the iterated format list is ignored.

END-AROUND REPETITIONS

The "end-around" repetition is a simple feature of the edit-directed statements. An outermost format list in an edit-directed statement is repeated when the end of the list has been reached. An outermost list is the format list paired with a data list in the statement. The effect of this convention is that the format list can never "run out" before the corresponding data list does.

The program fragment that follows shows the use of the compound format items:

```
        ...
F5:     FORMAT(A(10), 2 P"BBB--9V.99");
        ...
        I = 5;
        PUT(...)(PAGE, (I-2)(SKIP, R(F5), COL(20), A));
        ...
```

The format list in the PUT statement is equivalent to the following format list:

```
PAGE,
SKIP, A(10), P"BBB--9V.99", P"BBB--9V.99", COL(20), A,
SKIP, A(10), P"BBB--9V.99", P"BBB--9V.99", COL(20), A,
SKIP, A(10), P"BBB--9V.99", P"BBB--9V.99", COL(20), A,
PAGE,
SKIP, A(10), P"BBB--9V.99", P"BBB--9V.99", COL(20), A,
... and so on, ad infinitum.
```

In this format list, the data format items have been underlined to distinguish them from control format items. It is the data format item that is matched with each item in the data list, so the portion of the format list shown would accommodate 16 items from the data list.

The edit-directed statement uses the same form of data list as the list-directed statement; and the interpretation of that list to produce a simple list of data items was given in the discussion of list-directed input/output. Now, immediately above, the interpretation of a format list to produce a simple list of format items has been given. On the basis of these interpretations, the items of any data list can be matched to the items of the corresponding format list.

Guidelines for Edit-Directed Input/Output

Edit-directed input/output is preferred whenever the programmer wants to assume control over the format of input or output. It provides a wide variety of facilities for specifying format; and even within edit-directed techniques there is a range of control over details. At one extreme, the programmer can use the fixed-point format item and require, in a rather indefinite way, that an optionally-signed constant appear in certain columns of a line. At the other extreme the programmer can use a picture and control the contents of a line on a character-by-character basis.

The format list associated with an edit-directed statement can easily become complicated and unintelligible. It is important that a layout diagram be made of the document being read or written, and that the format-list be based on this diagram. The FORMAT statement can be used to structure a complicated format list just as the procedure is used to structure a complicated program.

The three disciplines of stream input/output can be mixed. For example, certain codes at the beginning of an input stream could be read by an edit-directed GET statement, and then a specified number of values could be read by a simple list-directed statement. However, care must be taken when switching back to edit-directed input. Since edit-directed input works on a strict column by column basis, a serious error can occur if the effect of the preceding non-edit-directed statement is not correctly determined.

THE STRING OPTION

In order to interpret the input/output statements, PL/I must have a large and complicated collection of string manipulation operations. In particular, the process of applying a format list to the input stream to produce values or of applying a format list to a value to produce an output stream is a complicated operation. Accordingly, PL/I has a facility to make this string manipulation available independent of the performance of input/output.

A GET statement can have an option of the form STRING(e) instead of the usual file option, where e is any character-string expression. In this case, the statement will take its input from the value of e as if that value were a complete stream data set. Similarly, a PUT statement can have an option of the form STRING(t) instead of the file option, where t is any target that can accept a character-string value. In this case, the statement will assign its entire output to t as if that target were a stream data set.

Linemarks and pagemarks cannot be used when the STRING option is used. If a GET statement with a STRING option "runs off the end" of the pseudo stream, the ERROR condition occurs rather than the ENDFILE condition. Thus the extension of input/output statements to the use of the STRING option applies only to the editing process itself and not to those aspects that are oriented toward input/output.

A useful application of the string option arises in connection with a troublesome property of stream input: the input of characters cannot be controlled by anything that appears later in the stream. Consider an example of this problem. Suppose 80-character card-images are being read and they can occur in either of two formats depending on whether an '*' or a blank appears in column 80. This problem can be solved by using the following statements:

```
....
GET EDIT(TEMP)(A(80));
IF SUBSTR(TEMP,80,1) = "*"
   THEN GET STRING(TEMP) EDIT(C1, C2, C3)(P"$$$$$V.99DB", X(7));
   ELSE GET STRING(TEMP) EDIT(C1, C2, C3)(P"$$$$$V.99-", X(8));
...
```

If a card ends with '*', this sequence of statements is equivalent to

```
GET EDIT(C1, C2, C3)(P"$$$$$V.99DB");
```

and otherwise the sequence is equivalent to

```
GET EDIT(C1, C2, C3)(P"$$$$$V.99-");
```

The use of the string option allows the program to "look ahead" in the input stream and select a format appropriate to the coming values.

CONDITIONS FOR STREAM INPUT/OUTPUT

In the following discussion, the conditions that occur during stream input/output are described. They are:

CONVERSION(ref)
ENDFILE(ref)
ENDPAGE(ref)
NAME(ref)
TRANSMIT(ref)
UNDEFINEDFILE(ref)

where ref is a reference that yields a file value. The general rules for the use of the conditions are given earlier, in the section on "Condition Handling". Only some remarks about their application to input/output will be given here.

Each condition is defined separately for each file value, and thus for each file-state block. The identifier ENDPAGE by itself is not a valid condition; but if RECORD3 is a file constant name, then ENDPAGE(RECORD3) is a valid condition. Consider the statement

```
ON ENDPAGE(RECORD3) PUT FILE(RECORD3) PAGE LINE(3);
```

When this statement is executed, it establishes the PUT statement as the ON unit for the condition ENDPAGE(RECORD3). When the end of a page in the output stream associated with RECORD3 is reached, the ENDPAGE is signalled and the ON unit is executed. When the block that contains the ON statement is deactivated, the ON unit is reverted, and no longer responds to a signal.

When a condition is signalled, the PL/I processor takes either of two actions, as follows:

- If an ON unit is established for the condition, then that ON unit is executed. If the execution of the ON unit runs to completion, then control goes back to the point in the program at which the condition occurred, and execution is resumed in a reasonable way (depending on the particular needs of the statement involved).

- If no ON unit is established for the condition, then the default ON unit is executed. The default ON unit for each condition is described earlier, in the section on "Condition Handling".

A stream input/output statement can evaluate expressions, and during that process a FIXEDOVERFLOW, OVERFLOW, UNDERFLOW, or ZERODIVIDE condition may occur. Further, a stream input statement assigns values to targets, and during that process a SIZE, STRINGRANGE, STRINGSIZE, or SUBSCRIPTRANGE condition may occur.

The PL/I processor saves certain useful values before signalling a condition. For each kind of value saved, there is a stack and a built-in function. Just before the condition is signalled, the value is placed on the top of the stack, and after completion of the established ON unit it is removed. The built-in function is used to access the value during the execution of the ON unit.

For example, just before any of the conditions mentioned in this section is signalled, the file name, expressed as a character-string value, is placed at the top of the stack controlled by the ONFILE() built-in function. During the execution of the established ON unit for the condition, the file-name character-string can be accessed by using the reference ONFILE(). When execution of the ON unit is complete, the file name character string is removed from the stack.

The CONVERSION Condition

The CONVERSION condition occurs when an attempt is made to convert an invalid character string or pictured value to an arithmetic or bit-string value. Just before the condition is signalled, three values are saved in the stacks controlled by the condition built-in functions. The character string being converted is placed at the top of the stack controlled by ONSOURCE(). The leftmost character in the string at which conversion failed, which is sometimes the source of the error, is placed at the top of the stack by ONCHAR(). The file name is saved as described in the preceding paragraph.

The ONSOURCE() and ONCHAR() functions can be used as pseudo-variables, and the ON unit can assign new variables to them; in this way, it is possible to "correct" a character string that is causing trouble. When a normal return from the ON unit occurs, the PL/I processor resumes its attempt to convert the offending character string. If the program has supplied a new and valid value by means of ONSOURCE() or ONCHAR() then the conversion succeeds, and execution continues; otherwise, the CONVERSION error occurs again.

The ENDFILE Condition

The ENDFILE condition occurs when an input statement attempts to read beyond the end of a data set. After an established ON unit is executed, the PL/I processor resumes with the statement after the input statement in which the condition occurred. If a later attempt is made to read the data set, the condition will occur again. The file name is saved in the stack controlled by ONFILE().

The ENDPAGE Condition

The ENDPAGE condition occurs when an output statement completes the nth line of a page (by writing a linemark) and n is equal to the "page size" associated with the output file. The condition can be caused in either of two ways, and the action taken by the PL/I processor on return from an established ON unit varies accordingly. If the condition was caused by an attempt to write a data value in the output stream, the output of the data is completed when execution resumes. But if the condition was caused by the interpretation of a SKIP option or format item or a LINE option or format item, then the option or format item is ignored; it is assumed that the ON unit starts a new page and eliminates the need for the blank lines.

When the ENDPAGE condition is signalled, the line number associated with the file has already been increased by one and is therefore equal to the PAGE SIZE plus one. Normally, the ON unit will include a PAGE option or format item and will thereby set the line number back to 1.

Just before the ENDPAGE condition is signalled, the file name is saved in the stack controlled by the ONFILE(). If there is no established ON unit for the condition, the PL/I processor does not treat the condition as an error; instead, a pagemark is added to the output stream, the line number is set to 1, and execution of the program continues.

The NAME Condition

The NAME condition occurs only during data-directed input. Specifically, the condition occurs when a stream assignment is read whose variable name does not match a variable name in the data list of the controlling GET statement or a name of a component of a variable that is named in the data list. Just before the condition is signalled, the offending assignment from the stream is placed at the top of the stack controlled by ONFIELD(), and the variable is therefore available as a character-string value for inspection with the ON unit. The file name is placed at the top of the stack controlled by ONFILE(). After an established ON unit is executed, the PL/I processor returns to the data-directed input as if the processing of the offending stream assignment were complete.

The TRANSMIT Condition

The TRANSMIT condition occurs when data cannot be transmitted reliably between a data set and PL/I storage. Just before the condition is signalled, the file name is placed at the top of the stack controlled by ONFILE(). After an established ON unit is executed, the PL/I processor resumes with the statement that follows the input/output statement that caused the condition; but the value of the data transmitted by the statement is undefined.

The condition is usually caused by factors beyond the programmer's control, such as a hardware failure, so the recovery procedure cannot be initiated until the hardware is repaired.

The UNDEFINEDFILE Condition

The UNDEFINEDFILE condition occurs when an OPEN statement attempts unsuccessfully to open a file. The condition can occur, when, for example, an attempt is made to open a record data set for stream input, or when, for another example, the TITLE option specifies a nonexistent file. Just before the condition is signalled, the file name is placed at the top of the stack controlled by ONFILE(). After the established ON unit is executed, the program resumes execution at the statement following the offending OPEN statement.

SECTION XV

RECORD INPUT/OUTPUT

The record input/output facility of PL/I is independent of the stream input/output facility described in the preceding section; that is, it has its own data sets, statements, and programming techniques. The record input/output facility is oriented toward communication with permanent storage. The role of such storage is to accept values from PL/I at one time and then return them, unchanged, at a later time; therefore, each value is transmitted just as it is found in PL/I storage. In contrast, stream input/output is oriented toward user communication and has many ways of converting between internal values and external representations of those values.

This section begins with a description of the two kinds of data that are involved in record input/output: the record data set, which is the actual subject of the input or output, and the file-state block, which shows the status of the operations on the record data set. In order to make this section complete and independent, the description of the file-state block repeats some material already given in the preceding section on "Stream Input/Output". The section continues by giving a summary of the operations that are performed as a part of record input/output. Once this foundation has been established, the section proceeds to a definition of the statements that are used for record input/output: first, the statements that open and close files and then the statements that perform the actual input/output operations. Next, the section describes based input/output, which is an advanced and specialized feature of record input/output. Finally, the section describes the conditions that occur in connection with record input/output.

RECORD DATA SETS

A record data set is a collection of records. Each record is a single PL/I value; that is, it is a copy of a value that once existed in PL/I storage. The record can be a single scalar value; indeed, it can be a "BIT(1)" value and thus represent only one bit. On the other hand thus, the record can be an aggregate value such as a large and complicated structure or an array of many elements. Some of the costs of transmitting and storing a record are the same for records of all sizes; therefore, large records are preferred. For example, if a programmer has a choice between treating an array as a single record or treating each element of the array as a record, then he should choose the first alternative.

The word "record" is used here to mean a logical record; that is, a collection of information gathered together because it belongs together. Hardware storage devices do have physical records; that is, units that reflect the architecture of the storage device. The relation of the PL/I logical record to the physical record is similar to the relation of the PL/I variable to the hardware computer word. In both cases, PL/I provides an elaborate and effective mechanism to allow a programmer to choose units that correspond to the logical requirements of the data and to ignore the boundaries that are built into the hardware.

The Organization of Record Data Sets

A record data set can be keyed, sequential, or keyed sequential. In a keyed data set, each record has a unique key associated with it that can be used to access the record directly without scanning through the file. In GCOS PL/I, the key is a character-string value of length up to 32 characters. In a sequential data set, the records are arranged in an order that does not change and that can be used to pass from one record to the next when the file is being processed. In a keyed sequential data set, a record can be accessed either by its key or by its sequential position. The organization of a data set determines the kinds of operations that can be performed on it.

When a file is being operated on, it has two indicators associated with it. The current record indicator designates the record that has been most recently operated on. The next record indicator designates the record that will be read if the next operation is a sequential read operation; it is defined only for a sequential file. Whenever the current record indicator is reset, the next record indicator is adjusted to designate the next record in sequence. Under certain circumstances, an indicator is set to null (and does not point to any record; for example, when current record indicator is set to designate the last record of a file, the next record indicator becomes null.

GCOS Files

There are a variety of ways to implement a record data set, each reflecting different hardware requirements and software techniques. GCOS PL/I has three implementations for a record data set: CONSECUTIVE, INDEXED, and REGIONAL files.

A CONSECUTIVE file can be used for an unkeyed sequential PL/I data set. Its records are arranged in the order in which they are created. A CONSECUTIVE file is either on a direct access device or on a magnetic tape.

An INDEXED file can be used for any keyed PL/I data set, sequential or not. Its records are arranged in order of ascending keys. That is, if the key k1 precedes the key k2 in a file, then the relation k1 < k2 (as defined for PL/I character strings) must be true. A record can be rewritten in any way; that is the storage type of the new record need not conform to that of the old record. An INDEXED file is always stored on a direct access device.

A REGIONAL file can be used for any keyed PL/I data set. A REGIONAL file consists of a number of regions, corresponding to the fixed length logical records of the file. A REGIONAL file is always stored on a direct access device.

RECORD FILES

A connection must be established between a statement that performs input/output and the GCOS file on which the operation is to be performed. An analysis of this connection follows:

- The connection begins with the file option that appears in an input/output statement.
- The file option has as its argument a file reference, and the evaluation of the file reference yields a file value.
- The file value designates a file-state block, which is a set of values that are used by the PL/I processor in carrying out input/output operations.
- The file-state block contains a file code that designates a GCOS file and thus completes the connection between input/output statement and file.

The main components in the connection just described are the file-state block and the file reference; these components are described in the following paragraphs.

First, however, a problem of terminology must be resolved. In PL/I, the source of input and the destination of output is called a data set; but in GCOS, it is called a file. This difference is observed when it is necessary to distinguish between the PL/I view of input/output, as in "a keyed sequential data set", and the GCOS view, as in "an indexed sequential file". The word "file" is also used as a PL/I term, and in that usage, it refers to the combination of the file-state block and the data set; thus, the phrase "open a file" actually refers to the setting of a certain file-state block to control input/output with a certain data set.

File-State Blocks

Transmission of values between the PL/I processor and a GCOS file requires bookkeeping data. This data is stored in a portion of system storage called a file-state block. When a file is open, the file-state block contains the GCOS file code and other information about input/output in progress. After the file is closed, the only information in the file-state block that is meaningful is that supplied by the attributes, if any, in the declaration of the file constant name. A file-state block cannot be accessed directly, but its values are changed when input/output is performed on the data set with which it is associated.

The following values in a file-state block are relevant to record input/output:

- The status indicator. This value shows whether the file-state block is open or closed.

- The GCOS file code. This value designates a GCOS file that is associated with the file-state block.

- The file name. This value is a character string that is the identifier that is the name of the constant file value that designates the file-state block.

- The file attributes. These attributes are those associated with the current use of the file-state block.

- The current record and next record indicators. These values point to the current position of input/output operations within the given data set.

In addition to the items just listed, there are other items, such as buffers, that are not of immediate interest to a programmer.

File References

A file-state block is designated by a file value, and the file value is supplied by a file reference in a FILE option. The file reference can be a reference to a constant, a variable, or a function.

A file constant reference is a name that has been declared with the following attributes:

```
[EXTERNAL]
[        ] FILE  [CONSTANT]
[INTERNAL]
```

The default rules provide that the scope attribute can be omitted if it is EXTERNAL. The CONSTANT attribute can be omitted in any case, as indicated by the square brackets.

Every file constant name must have an associated file description. This file description may be given when the file is opened, as described later in this section, under "The OPEN Statement". However, PL/I also allows the programmer to write any portion of the file description attributes in the declaration of the file constant name.

Each declaration of a file constant name associates the name with its own file-state block in static system storage. The only exception is the declaration of a given name in several different blocks as EXTERNAL FILE CONSTANT; in this case, the declarations all refer to a single file-state block, as is required by the interpretation of the EXTERNAL attribute. A given file constant name and its associated file-state block can be used for more than one data set in the course of execution of a program. For example, a file-state block can be opened for input from a stream data set, closed, opened for updating a record data set, closed again, and so on.

A file variable reference or a file function reference is similar to a variable reference or a function reference of any other type. However, two exceptional features of file variable names are:

- The default scope of a FILE variable name is EXTERNAL, whereas the default scope for most other variable names is INTERNAL.
- The attribute VARIABLE must be used explicitly for a file variable name because the default for a name of type FILE is CONSTANT.

RECORD INPUT/OUTPUT OPERATIONS

A summary of record input/output operations is given here. It introduces terminology, shows how data sets are manipulated, and gives a general view of the record input/output facility.

When a data set is opened for output, the contents of the data set are discarded and the data set is ready to accommodate the writing of new records. When a data set is opened for input, the contents are retained and the data set is made available for reading of its records. When a data set is opened for update, the contents are retained and the data set is made available for reading, writing, deleting, or rewriting of records. When a sequential data set is opened for INPUT or UPDATE, the next record indicator is set to designate the first record of the data set. The contents of a data set remain accessible to PL/I in this way until the data set is closed.

A given input/output operation uses either the keyed or the sequential properties of a data set, but not both; and this distinction is useful in the description of record input/output. A keyed operation uses the key supplied by an input/output statement to find the record to be operated on. A sequential operation uses the current record or next record indicators for this purpose.

When a record is created and assigned a value it is said to have been written. A keyed write operation places the new record in its proper sequential position to maintain the ascending sequence of keys. An unkeyed write operation places the new record at the end of the data set. In either case, the value is copied into the record exactly as it appears in the referenced variable in PL/I storage.

A keyed read operation begins by locating the record that has the specified key and designating it as the current record. A sequential read operation begins by designating the next record as the current record (and thus advancing by one record). In either case, the value of the current record is then copied into the designated unit of PL/I storage. If the storage type of the record and the storage unit are not identical, the operation is invalid.

A keyed delete operation begins by locating the record that has the specified key and designating it as the current record. A sequential delete operation begins by finding the current record. In either case, the current record is then discarded, with the result that there no longer is a current record; that is, the current record indicator is set to null.

A rewrite operation replaces an existing record with a new record. The new record must be of the same size and key value as the old record.

The based input/output operations are a relatively specialized facility. When based input is performed, PL/I automatically allocates storage with storage type identical to the record; and thus a record can be input even when its storage type cannot be predicted by the programmer.

A broad spectrum of errors can occur during record input/output. An attempt to modify a record in a data set that was opened for output is a programming error. The input of a record whose structural attributes do not agree with the designated PL/I storage unit may be an input-data error. Inaccurate transmission of a value between a data set and PL/I storage is a system error. And finally, an attempt to read beyond the end of a sequential data set may not be an error at all but rather a convenient way of ending an input loop. PL/I detects these conditions when they occur and the programmer can provide an ON unit to respond to each condition with suitable actions.

OPENING AND CLOSING FILES

When a file is opened, the file-state block is marked "open" and the data set designator and control parameters and indexes are set in the block. When a file is closed, the file is marked "closed" and only information provided by the file declaration is meaningful.

A file is opened when the first input/output statement referencing the file is executed. The purpose of the OPEN statement is to provide the title and file description for the file opening. However, both these options can be omitted from the OPEN statement, and, in that case, a default assumption is made. If an OPEN statement is not given for a file, the attributes for the file opening are derived from the first input/output statement executed. If a file is already open when an OPEN statement is executed, the OPEN statement is completely ignored.

The OPEN Statement

An OPEN statement gives a file value, a title, and a file description. Consider the statement

```
OPEN FILE(SUBSCRIBER) TITLE("X1") KEYED SEQUENTIAL UPDATE;
```

In this statement, the file value is given by the file constant name SUBSCRIBER, the title specifies the file code X1, and the file description is KEYED SEQUENTIAL UPDATE. The statement is interpreted as follows:

- The title is used to designate a GCOS file, and that file is associated with the given file constant.
- The attached GCOS file is checked to see that it conforms to the file description. For this statement, the file must be INDEXED and must be available for both reading and writing.
- The current record indicator and the next record indicator are set to the first record of the data set.
- Finally, the file-state block is marked "open".

When a data set is opened for OUTPUT, the effect is to create a new data set. For example, the statement

```
OPEN FILE(SUBSCRIBER) TITLE("X1") KEYED SEQUENTIAL OUTPUT;
```

has quite a different effect than the previous example. This statement deletes the data set designated by X1 and creates a new, empty data set whose organization conforms to the file description.

The TITLE option can be omitted from an OPEN statement, in which case the file constant name is used as the title. For example, the statement

```
OPEN FILE(SUBSCRIBER) KEYED DIRECT OUTPUT;
```

is equivalent to

```
OPEN FILE(SUBSCRIBER) TITLE ("SUBSCRIBER") KEYED DIRECT OUTPUT;
```

and the character string "SUBSCRIBER" is used to attach a file to the file constant designated by SUBSCRIBER.

FILE DESCRIPTIONS

The following diagram gives every complete file description that can be used to open a data set for record input/output:

```
{SEQUENTIAL    }{INPUT  } [ENVIRONMENT({CONSECUTIVE})] [RECORD]
{KEYED SEQUENTIAL}{OUTPUT } [          {INTERACTIVE}]
{KEYED DIRECT  }{UPDATE } [          {INDEXED    }]
                         [          {REGIONAL   }]
```

A given file description specifies a processing mode, the type of input/output operations to be performed, and the file organization. The ENVIRONMENT attribute can be omitted and, in this case, the file organization is determined at execution either by control cards or by default. The RECORD attribute can also be omitted. There are other rules for shortening the file description, but they are complicated and their use is not recommended.

During the time a data set is open under a given file description, the attributes in that file description determine which input/output operations are permitted. The attributes with which the file description begins determine whether the operations can be keyed, sequential, or both, as follows:

KEYED DIRECT	permits keyed operations only
SEQUENTIAL	permits sequential operations only
KEYED SEQUENTIAL	permits both keyed and sequential operations

The attribute with which the file description continues determines the kind of statement that can be used to perform input/output, as follows:

OUTPUT	permits the use of a WRITE or LOCATE statement only
INPUT	permits the use of a READ statement only
UPDATE	permits the use of a WRITE, READ, DELETE, or REWRITE statement only; however, a WRITE statement is permitted only if the file description includes the KEYED attribute

Observe that UPDATE permits almost any input/output statement; however, it does not permit the use of a LOCATE statement (which is rarely used in any case) or the use of a WRITE statement for a data set that is opened as unkeyed SEQUENTIAL.

The ENVIRONMENT attribute specifies the file organization. The file organization determines the method for storing and retrieving records. SEQUENTIAL INPUT and OUTPUT can be performed on any file, but SEQUENTIAL UPDATE cannot be performed on a file with INTERACTIVE organization. KEYED INPUT and UPDATE can be performed on INDEXED or REGIONAL files, but KEYED OUTPUT can be performed only on files with REGIONAL organization. A detailed description of the different types of file organization is given in the GCOS PL/I User's Guide.

The CLOSE Statement

The close statement has a simple form, as indicated by the following example:

```
CLOSE FILE(SUBSCRIBER);
```

This statement marks the file-state block SUBSCRIBER closed. In addition, it clears and frees any buffers which have been allocated and set by previous based input/output operations. These buffers are discussed under "Based Input/Output", later in this section.

KEYED INPUT/OUTPUT OPERATIONS

When an input/output statement contains a KEY or a KEYFROM option, it performs keyed input/output. The file on which such a statement operates can usually be either DIRECT or SEQUENTIAL, but it must be KEYED in any case.

The Keyed WRITE Statement

Consider the statement:

```
WRITE FILE(EMPLOYEE) KEYFROM(SSNO) FROM(ITEM(3));
```

The file EMPLOYEE must be a KEYED OUTPUT or KEYED UPDATE file. The statement attempts to create a new record in the file EMPLOYEE under the key given by the value of the character-string variable named SSNO. If the file is KEYED SEQUENTIAL OUTPUT, the key given by SSNO must be greater than any key already in the file (so the record goes at the end of the file); otherwise, the KEY condition occurs. The created record becomes the current record and its value is the current value of ITEM(3). However, the operation fails and the KEY(EMPLOYEE) condition occurs if there is already a record in EMPLOYEE under the key given by SSNO.

The Keyed READ Statement

The READ statement with the KEY option is used for keyed input from a file. Consider the statement:

```
READ FILE(EMPLOYEE) KEY(SSNO) INTO(REC.MAIN);
```

The file EMPLOYEE must be a KEYED INPUT or KEYED UPDATE file. The statement attempts to find a record in the file EMPLOYEE that has the key given by SSNO. If such a record is found, it becomes the current record and is read into the PL/I storage designated by REC.MAIN.

The reference in the INTO option must not be a VARYING string variable reference. The operation fails and the KEY(EMPLOYEE) condition occurs if there is no record in EMPLOYEE under the key given by SSNO.

Input and output values must be matched exactly. Suppose the following statements are executed in sequence:

```
WRITE FILE(EMPLOYEE) KEYFROM(SSNO) FROM(ITEM(3));

READ FILE(EMPLOYEE) KEY(SSNO) INTO(REC.MAIN);
```

If ITEM(3) and REC.MAIN have exactly the same structural attributes, these statements are equivalent to:

```
WRITE FILE(EMPLOYEE) KEYFROM(SSNO) FROM(ITEM(3));

REC.MAIN = ITEM(3);
```

The equivalence just given breaks down when the structural attributes of ITEM(3) and REC.MAIN are not exactly the same. When data types do not match in an input/output statement, PL/I does not convert the value, and when aggregate types do not match PL/I does not attempt to promote. Instead, any disagreement of structural attributes is an error.

The Keyed DELETE Statement

The DELETE statement with the KEY option is used for the keyed deletion of an existing record from a file. Consider the statement:

```
DELETE FILE(EMPLOYEE) KEY(SSNO);
```

The file EMPLOYEE must be a KEYED UPDATE file. This statement attempts to delete a record from the file EMPLOYEE under the key given by SSNO. The key is deleted from the file as well as the record, so the key is unused in this file after the delete operation. The KEY(EMPLOYEE) condition occurs if there is no record in EMPLOYEE with the key given by SSNO.

The Keyed REWRITE Statement

The REWRITE statement with the KEY option is used to write a new version of an existing record in a keyed file. Consider the statement:

```
REWRITE FILE(EMPLOYEE) KEY(SSNO) FROM(CORRECTION);
```

The file must be a KEYED UPDATE file. This statement attempts to output a record to the file EMPLOYEE and enter it under the key given by SSNO. The new value of the record is taken from the variable CORRECTION, and the old value of the record is destroyed. The KEY(EMPLOYEE) condition occurs if there is no record in EMPLOYEE under the key given by SSNO.

Since the WRITE statement uses the KEYFROM option to specify the key, the programmer may be tempted to use a KEYFROM option in the REWRITE statement; but this is a syntactic error. In PL/I, the KEY option is used when a statement attempts to find a given key in a file (as in the READ, DELETE, and REWRITE statements), and the KEYFROM option is used when a statement attempts to introduce a given key into the file (as in the WRITE statement).

An Example of Keyed Input/Output

Suppose a simple list of subscribers to a monthly magazine is stored as a keyed sequential file. Each record gives the name and address of a subscriber and the date of expiration of his subscription. The key for each record is a 12-character string which is made up of the zip code and other identifying information. The problem is to extend the date of expiration of subscribers as their subscriptions are renewed. The program is as follows:

```
RENEW:  PROC;
           DCL SYSIN FILE;
           DCL GIVEN FILE:
           DCL SKEY CHAR(12) VAR;
           DCL 01 SUBS,
                  02 NAME CHAR(30) VAR,
                  02 ADDRESS CHAR(60) VAR,
                  02 EXPIRY,
                     03 MONTH DEC(2),
                     03 YEAR DEC(2);
           OPEN FILE(GIVEN) DIRECT UPDATE;
           DO WHILE ("1"B);
              GET LIST(SKEY);
              IF SKEY = "END"
                 THEN DO; CLOSE FILE(GIVEN); RETURN; END;
              READ FILE(GIVEN) KEY(SKEY) INTO(SUBS);
              YEAR = YEAR+1;
              REWRITE FILE(GIVEN) KEY(SKEY) FROM(SUBS);
              END;
          END;
```

An example of input for the program is

```
"94305MARSA82"
"02139STEIS95"
"20742MARTB61"
"END"
```

This procedure reads keys from the input stream SYSIN and adds 1 to the year of expiration for the corresponding record. The fact that the subscription file is declared to be DIRECT does not imply that the data set is not sequential; it only means that this use of the data set will not depend on whether or not it is sequential. Indeed, a later example in this section uses this same file for sequential input/output.

SEQUENTIAL INPUT/OUTPUT OPERATIONS

When an input/output statement does not contain a KEY or a KEYFROM option, it performs sequential input/output. The file on which such a statement operates must have the SEQUENTIAL attribute.

The Sequential WRITE Statement

Consider the statement:

```
WRITE FILE(SUBSCRIBER) FROM(CUST);
```

The file SUBSCRIBER must be an unkeyed SEQUENTIAL OUTPUT file. The statement creates a new record at the end of the file, and the current value of CUST is assigned to the record.

The Sequential READ Statement

The READ statement without the KEY option is used for sequential input from a file. Consider the statements:

```
READ FILE(SUBSCRIBER) INTO(CUST);
```

and

```
READ FILE(EMPLOYEE) KEYTO(SSNO) INTO(CUST);
```

The file SUBSCRIBER must be a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file (keyed or not), and the file EMPLOYEE must be a KEYED SEQUENTIAL INPUT or KEYED SEQUENTIAL UPDATE file. The indicator associated with the file is moved to the next record; that is, the current record indicator in the file-state block is given the value of the next record indicator. Then the value of the new current record is assigned to CUST in PL/I storage. The KEYTO(SSNO) option causes the key associated with the current record to be assigned to SSNO in PL/I storage.

The reference in the INTO option must not be a VARYING string variable reference.

The READ statement can also be used to skip over records in a sequential file. Consider the statement:

```
READ FILE(SUBSCRIBER) IGNORE(3);
```

The file SUBSCRIBER must be a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file (keyed or not). If the current record indicator designates the ith record of the file, then IGNORE(3) moves the indicator to the (i+3)th record. If the file has just been opened, the current record indicator is set to null; but, for this calculation, it can be thought of as designating the zeroth record. If the end of the file is reached before the operation is complete, the current record indicator is set to the null record and the ENDFILE(SUBSCRIBER) condition is signalled. The argument of the IGNORE option must be greater than zero.

The Sequential DELETE Statement

The DELETE statement without the KEY option is used for deletion of the current record of the file. Consider the statement:

```
DELETE FILE(SUBSCRIBER);
```

The file SUBSCRIBER must be a SEQUENTIAL UPDATE file (keyed or not). The statement causes the current record to be deleted; that is, it sets the current record indicator to null. Since the next record indicator is set to the following record, the statement does not cause PL/I to "lose its place" in the file; but the next input/output operation on the file, if any, must be a keyed operation or a sequential read so that a new record is selected.

The Sequential REWRITE Statement

The REWRITE statement without the KEY option is used to write a new version of an existing record in a sequential file. Consider the statement:

```
REWRITE FILE(SUBSCRIBER) FROM(RENEWAL);
```

The file must be SEQUENTIAL UPDATE. The statement replaces the contents of the current record with the value of RENEWAL in PL/I storage; and the old value of the record is destroyed.

An Example of Sequential Input/Output

Once again a list of subscribers to a monthly magazine is stored as a keyed sequential file. The problem is to read through the file sequentially, checking each record in turn to see if the subscription has run out. Those records that represent expired subscriptions are copied into another keyed sequential file. The program is as follows:

```
EXPTEST:  PROC;
             DCL GIVEN FILE;
             DCL TARDY FILE;
             DCL SKEY CHAR(12);
             DCL 01 SUBS,
                    02 NAME CHAR(30),
                    02 ADDRESS CHAR(60),
                    02 EXPIRY,
                       03 MONTH DEC(2),
                       03 YEAR DEC(2);
          OPEN FILE(GIVEN)  KEYED SEQUENTIAL INPUT;
          OPEN FILE(TARDY)  KEYED SEQUENTIAL OUTPUT;
          ON ENDFILE(GIVEN) GOTO EXIT;
          DO WHILE("1"B);
             READ FILE(GIVEN) KEYTO(SKEY) INTO(SUBS);
             IF 12*YEAR+MONTH < 12*74+3
             THEN WRITE FILE(TARDY) KEYFROM(SKEY) FROM(SUBS);
             END;
EXIT:     CLOSE FILE(TARDY);
          CLOSE FILE(GIVEN);
          END;
```

The program tests for subscriptions that expired before March 74. Its most interesting point, however, is the use of KEYED SEQUENTIAL output. These attributes require that TARDY be written in order of ascending keys; and this requirement is satisfied, since the program is copying from a file, GIVEN, that is KEYED SEQUENTIAL and necessarily satisfies the requirement. The program would still be valid if TARDY were declared KEYED DIRECT, but the useful fact that the records will be written in order of ascending keys would not be made explicit and PL/I might not perform the output as efficiently as possible. Therefore the declaration KEYED SEQUENTIAL is best.

BASED INPUT/OUTPUT OPERATIONS

Based input/output is a rather advanced and difficult technique of PL/I programming. Fortunately, based <u>output</u> is not important in Honeywell PL/I and need be given no more than passing mention at the end of this discussion. However, based <u>input</u> is useful, especially in commercial programming.

Consider a programming application in which the following vicious circle arises:

The records of a given input file are in several different structural forms, and therefore a particular record cannot be read into PL/I storage until the storage type of its values have been determined. However, the records occur in an unpredictable order, and the only indication of the storage type of a particular record is a code that is contained within the record. In short, the record cannot be read until its form is known and its form cannot be known until the record has been read.

The based input statement breaks this circle by reading a record into system storage and thus using a special technique not otherwise available to the user.

Based Input

A based input statement can be obtained by writing a READ statement with a SET option instead of an INTO option. Three forms are possible, as the following examples show:

```
READ FILE(LOG) KEY(ITEMNO) SET(PTR);

READ FILE(LOG) SET(PTR);

READ FILE(LOG) KEYTO(ITEMNO) SET(PTR);
```

The argument of the SET option must be a target for a POINTER value. In each of these statements, the record is located just as it would be for the statement with an INTO option; the first statement is a keyed operation, the second and third are sequential operations. When the record has been located, PL/I allocates enough storage from system storage to hold the value of the record, copies the record into the storage, and sets PTR to point to the beginning of the allocated storage.

It is useful to think of the input process as follows: PL/I examines the record, allocates storage with exactly the same structural attributes as the record, and then reads the record into that storage. This could not be done with an ordinary READ statement (with an INTO option); there is no way that a program can examine a record before reading it in, and an ordinary READ statement must specify the attributes of the target before the input is performed.

Once the record has been input by means of a based input statement, it is interpreted by using the pointer value to associate a based variable with the value of the record. Many techniques for the use of based variables can be used; but the most important ones are given in the following examples.

AN EXAMPLE OF BASED INPUT

In the first example, a file that contains the daily transactions of a repair shop must be read. There are different kinds of transactions, and therefore different kinds of structures are required to represent them. The transactions covered are as follows;

Code	Purpose
1	Order a replacement part, giving the part number and the source of supply.
2	Bill a customer, giving name, address, and amount due.
3	Record an internal charge, giving a cost center and a cost.

A program is required to read through the file accumulating the credits. A Code 1 transaction is ignored, but each Code 2 or Code 3 transaction contributes its AMOUNT_DUE or COST to the accumulated credits. When the file has been completely read, the accumulated total is printed.

If based input were not available, the file could be designed with two records for each transaction. The first record of any pair would give the transaction code and the second would be a structure describing the transaction. The program would read the code, choose the appropriate target for the second record, and then read the second record. The disadvantage of this approach is that it uses twice as many records as necessary, and in many cases, would nearly double the cost of both the storage and the processing.

Since based input <u>is</u> available, the file can be written with one record for each transaction. Each record is a structure whose first member is the transaction code and whose remaining members are appropriate to the kind of transaction described. When a record has been read, the code is examined (without looking at the rest of the record) and is used to select a statement that will use an appropriate based variable to interpret the value of the record. The program is as follows:

```
SUM:    PROC;
           DCL TOTAL PIC"$$$$$$.V99";
           DCL TRANS FILE;
           DCL SYSPRINT FILE;
           DCL P POINTER;
           DCL 01 ORDER BASED(P),
                  02 CODE DEC(1),
                  02 PART_NUMBER CHAR(12),
                  02 SUPPLIER DEC(3);
           DCL 01 BILL BASED(P),
                  02 CODE DEC(1),
                  02 CUSTOMER,
                     03 NAME CHAR(20) VAR,
                     03 ADDRESS CHAR(40) VAR,
                  02 AMOUNT_DUE PIC"$$$$.V99";
           DCL 01 CHARGE BASED(P),
                  02 CODE DEC(1),
                  02 COST_CENTER CHAR(3),
                  02 COST PIC"$$$$.V99";
           ON ENDFILE(TRANS) GOTO EXIT;
           OPEN FILE(TRANS) SEQUENTIAL INPUT;
           TOTAL  = 0;

LOOP:      READ FILE(TRANS) SET(P);
           GOTO L(ORDER.CODE);
L(1):      GOTO LOOP;
L(2):      TOTAL =    TOTAL + AMOUNT_DUE;
           GOTO LOOP;
L(3):      TOTAL =    TOTAL + COST;
           GOTO LOOP;
EXIT:      PUT SKIP LIST("TOTAL BILLING: ", TOTAL);
           CLOSE FILE(TRANS);
           END;
```

This program is easy to read, but it is not so easy to write. The application of based variables to the input value must be programmed carefully because errors may not be detected.

The declarations of the structures ORDER, BILL, AND CHARGE are as might be expected from the definition of the problem. The association of the pointer P with each of the three structures saves writing later. For example, ORDER.CODE means P->ORDER.CODE because ORDER is declared BASED(P).

The based input statement at LOOP reads the value of a record into system storage and sets P to the beginning of the value. The important statement is:

```
GOTO L(ORDER.CODE);
```

This statement evaluates the transaction code by overlaying the structure ORDER on the input value and then obtaining the value of ORDER.CODE. The value of ORDER.CODE must be 1, 2, or 3 and the appropriate transfer to L(1), L(2), or L(3) is performed.

Once the code has been examined, the correct choice for a based variable can be made. For example, at L(2), it is known that the record represents a bill for a customer, and therefore reference can be made to AMOUNT_DUE. The fully-qualified equivalent of this reference is P ->BILL.AMOUNT_DUE, and it means: treat the value pointed to by P as if it were like the structure BILL and get the value of AMOUNT_DUE from that structure.

ANOTHER EXAMPLE OF BASED INPUT

In this example, a program generates a two-dimensional array and writes it out as a one-record file. Later, another program reads the array and processes it. It is assumed that the programs are used repeatedly in this sequence, and that the array may have different extents from one application to the next; therefore, the extents cannot be given as constants. For these programs, self-describing structures are used; that is, each array is incorporated in a structure that contains integers that represent the bounds of the array. The self-describing structure is designed in a special way so that not only the programmer knows where the extents are stored, but the PL/I processor also knows. The program that outputs the array is, in part, as follows:

```
BUILD:  PROC;
            DCL (M,N) FIXED;
            DCL 01 S BASED(P),
                   02 EXTENTS,
                      03 E1 FIXED,
                      03 E2 FIXED,
                   02 A(M REFER(E1):N REFER(E2));
            DCL P PTR;
            DCL X FILE;

            ... (SET M AND N TO THE DESIRED VALUES) ...

            ALLOCATE S;

            ... (SET THE M*N VALUES OF THE ARRAY) ...

            OPEN FILE(X) SEQUENTIAL OUTPUT;
            WRITE FILE(X) FROM(S);
            CLOSE FILE(X);
            END;
```

The interesting action in this program is the ALLOCATE statement. This statement allocates the structure S in system storage using the current values of M and N as the bounds for the array. At the same time, the "allocate" statement assigns the bounds to E1 and E2, so that an explicit record of the bounds of the array is made. These are useful when the array is read in by the second program that follows:

```
USE:  PROC;
         DCL (M, N) FIXED;
         DCL P POINTER;
         DCL 01 S BASED(P),
                02 EXTENTS,
                   03 E1 FIXED,
                   03 E2 FIXED,
                02 A(M REFER(E1): N REFER(E2));
         DCL X FILE;
         OPEN FILE(X) SEQUENTIAL INPUT;
         READ FILE(X) SET(X);
         CLOSE FILE(X);

         ... (MAKE USE OF A)

         END;
```

Because the array is self-describing and uses REFER options in the proper way, PL/I "understands" that the extents of the array are given by E1 and E2. Incidentally, the variables M and N are not used at all in USE; they are given to cover the allocation of the array A, but the array is not allocated in this program. PL/I allocates storage according to the requirements of the input record, not the declaration of S; but then a reference to A is interpreted as if that storage had been allocated by the statement ALLOCATE S;.

Based Output

An ordinary output statement transfers a given value to a portion of system storage called an output buffer; then the actual output is performed under control of the operating system from that output buffer. There are certain techniques that permit a programmer to gain access to the output buffer, and these are described in the following paragraphs. These techniques should be used only when the need is clearly evident.

THE OMISSION OF THE FROM OPTION

It is possible to perform output directly from the input buffer; that is, under certain circumstances, the FROM option can be omitted from a REWRITE statement, as follows:

```
REWRITE FILE(EMPLOYEE) KEY(SSNO);
```

or

```
REWRITE FILE(SUBSCRIBER);
```

Such a statement is allowed only if the last input operation on EMPLOYEE or SUBSCRIBER, respectively, was a based READ operation. In that case, the REWRITE statement takes its output value from the input buffer associated with the file by the preceding READ statement. Thus it is possible to read the value of a record into a buffer, modify it in that buffer, and write the value out from the same buffer.

THE LOCATE STATEMENT

A second facility for programmer control of buffers is the LOCATE statement. The LOCATE statement is related to the WRITE statement as the based READ statement is related to an ordinary READ. Consider the statement:

```
LOCATE BUF SET(PTR) FILE(EMPLOYEE) KEYFROM(SSNO);
```

The BUF must be a BASED variable, and if it is declared BASED(PTR) then the SET option can be omitted. If the file is not keyed, the KEYFROM option must be omitted. The effect of this locate statement can be described in terms of replacement by two other statements. The LOCATE statement itself can be replaced by a statement to allocate BUF, as follows:

```
ALLOCATE BUF SET(PTR);
```

Then, at a later point in the program, just before the next output operation on the file EMPLOYEE or just before the closing of the file EMPLOYEE, the actions implied by the following statement are carried out:

```
WRITE FILE(EMPLOYEE) KEYFROM(SSNO) FROM(BUF);
```

Thus the LOCATE statement sets up an output buffer but the contents of the buffer are not written out until the last possible moment.

CONDITIONS FOR RECORD INPUT/OUTPUT

In this discussion, the conditions that occur during record input/output are described. They are:

```
ENDFILE(ref)
KEY(ref)
RECORD(ref)
TRANSMIT(ref)
UNDEFINEDFILE(ref)
```

where ref is a reference that yields a file value. The general rules for the use of conditions are given earlier, in the section on "Condition Handling"; only a summary is given here.

As indicated above, each condition is defined separately for each file constant, and thus for each file-state block, whether it is open or not. The identifier ENDFILE is not a valid condition; but if SUBSCRIBER is declared FILE, then ENDFILE(SUBSCRIBER) is valid.

As an example of condition handling, consider the statement:

```
ON ENDFILE(SUBSCRIBER) GOTO EXIT;
```

When this statement is executed, it establishes the ON unit GOTO EXIT; for the condition ENDFILE(SUBSCRIBER). If the ENDFILE(SUBSCRIBER) condition is signalled, the ON unit itself is executed. When the block that contains the ON statement is deactivated, the ON unit is reverted and no longer responds to a signal.

When a condition occurs, PL/I takes either of two actions, as follows:

- If an ON unit is established for the condition, then that ON unit is executed. If the execution of the ON unit runs to completion, control goes back to the point in the program at which the interruption occurred, and execution is resumed in a reasonable way (depending on the particular needs of the condition).

- If no ON unit is established for the condition, the default ON unit is executed. The default ON unit for each condition is described earlier, in the section on "Condition Handling".

PL/I saves certain values before signalling a condition. For each kind of value saved, there is a stack and a built-in function. Just before the condition is signalled, the value is placed on the top of the stack, and after completion of the established ON unit, it is removed. The built-in function is used to access the value during the execution of the ON unit.

When a record input/output operation causes a condition to be signalled, the following values are saved in the manner just described. First, the file name, expressed as a character-string value, is saved in the stack associated with the ONFILE() built-in function. Second, if the file being operated on has been opened with the attribute KEYED, then the current key is saved in the stack associated with the ONKEY() built-in function.

The ENDFILE CONDITION

Suppose the file SUBSCRIBER is positioned so that its current record is the last record in the file, and suppose a sequential READ statement is executed. Since there is no next record, PL/I signals ENDFILE(SUBSCRIBER). If an ON unit is established for ENDFILE(SUBSCRIBER), then it is executed; and if the ON unit runs to completion, execution of the program resumes with the statement after the READ statement that caused the condition to occur.

Sometimes the number of records in a file is known in advance, and that number can be used to control the loop that reads the records. In such a case, an ENDFILE condition indicates an error in the preparation of the input file. The programmer may choose to provide an ON unit for recovery from such an error or he may decide to accept the diagnostic message and program abort that the system supplies by default.

An occurrence of an end-of-file condition is not necessarily an error; indeed, it is an excellent way to terminate a loop that processes the records of a file. Several of the example programs given in this section use a statement such as the following:

```
ON ENDFILE(SUBSCRIBER) GOTO EXIT;
```

to exit from a loop that is reading the records of an input file. Such programming is especially elegant. A programmer writes a loop that would go on reading records forever if there were no end to the file. Then quite separately, the programmer writes an ON statement that determines the action to be taken when the end of the file is reached. This separation of activities makes the program easier to write and easier to understand.

The KEY Condition

The KEY(SUBSCRIBER) condition occurs when a wrong assumption is made about the keys in the file SUBSCRIBER. That is, it occurs when a keyed WRITE statement has a KEYFROM option which supplies a key which is already in the file; and it occurs when a keyed READ, DELETE, or REWRITE statement has a KEY option that supplies a key that is <u>not</u> already in the file. If an ON unit is established for the condition KEY(SUBSCRIBER) it is executed; and if it runs to completion, then execution continues with the statement after the input/output statement in which the condition occurred.

This condition has an important role in signalling errors in the use of a keyed file. It can also be used to support a legitimate inquiry about the use of a key in a file. For example, suppose a file of employees is keyed by social security numbers. Then a given number can be checked to see if its owner is an employee. First, the statement

```
ON KEY(EMPLOYEE) EMP = "0"B;
```

is executed. Then the following statements are used to branch according to whether or not the number was in the file:

```
EMP = "1"B;
READ FILE(EMPLOYEE) KEY(GIVEN_SSNO) INTO(INFO);
IF EMP THEN GOTO CONTINUE; ELSE GOTO NEXT;
```

The RECORD Condition

The RECORD(SUBSCRIBER) condition occurs when the value of a record from the file SUBSCRIBER does not fit into the storage provided by the INTO option of a READ statement. A value fits if the number of bits it requires is exactly the number of machine-level bits allocated in storage. This condition is implementation dependent; however, the following assertion is true for any implementation of PL/I:

> If the RECORD condition occurs, then the input value does not have the same storage type as the target given by the INTO option of the READ statement.

Such a condition indicates an error in the program or the input file. After an established ON unit is executed, PL/I completes its execution of the READ statement by assigning the value of the record to the target, truncating it or padding it with zero-valued bits to make it fit the target.

Although it is an error to input a value that does not have the same storage type as the target, this condition only occurs when the number of bytes required by the value of the record differs from the number of bytes in the target. Thus, certain errors go undetected.

The TRANSMIT Condition

The TRANSMIT(SUBSCRIBER) condition occurs when data cannot be reliably transmitted between the file SUBSCRIBER and the PL/I storage referenced in the statement that attempted the input or output. After an established ON unit is executed, the program resumes at the point following the input/output statement that caused the condition; but the value of the data transmitted by the statement is undefined.

The condition is usually caused by factors beyond the programmer's control, such as hardware failure, so the recovery procedure usually cannot be initiated until the hardware problem is resolved.

The UNDEFINEDFILE Condition

The UNDEFINEDFILE(SUBSCRIBER) condition occurs when an OPEN statement attempts unsuccessfully to open the file SUBSCRIBER. The condition can occur when, for example, an attempt is made to open an unkeyed data set for KEYED input or output, or when, for another example, the TITLE option specifies an illegal attachment. After an established ON unit is executed, the program resumes at the point following the OPEN clause. This point may be the next of a series of OPEN clauses in an OPEN statement or the statement after the OPEN statement.

APPENDIX A

GUIDE TO PL/I STATEMENTS

A brief description of each of the PL/I statements is given in this appendix. The emphasis is on the syntax of the statements rather than on the interpretation. The entries are arranged in alphabetical order according to the initial keyword of each statement.

PRELIMINARY REMARKS

The statement descriptions given in this appendix have three parts: the syntax diagram, the supplementary rules, and the brief interpretation. The syntax diagram is enclosed in a box and uses the special notation described later. In the description of the SIGNAL statement that appears in this appendix, the following syntax diagram is given:

```
SIGNAL  cd  ;
```

This diagram defines the SIGNAL statement as "the identifier SIGNAL, followed by a cd, followed by a ';'". The supplementary rules provide information about the statement that is not given in the diagram. For the SIGNAL statement, the supplementary rules are just the clause:

where cd must be a condition designator

This clause defines the syntactic variable, cd, so that the diagram can now be interpreted as "the identifier SIGNAL, followed by a condition designator, followed by a ';'". The brief interpretation discusses the action taken by the statement and refers the reader to the appropriate section(s) of this manual for a complete definition. For the SIGNAL statement, the brief interpretation is:

The statement signals the condition designated by cd. See the section on "Condition Handling".

The Syntax Notation

The following paragraphs are a complete description of the notation that is used in the syntax diagrams. Many examples are given.

BASIC CONSTRUCTS

In a syntax diagram, an underlined identifier is a syntax variable; it represents a set of constructs that is defined somewhere else, either in another diagram or in the text. Certain other parts of a diagram, described in the following paragraphs, are used for special purposes, such as indicating a list of items. All the remaining characters in the syntax diagram are literal constructs. Thus in the diagram already given for the SIGNAL statement, the underlined identifier 'cd' is a syntax variable and SIGNAL and ';' are taken literally.

The syntax variables and literal constructs of a diagram are separated from one another by blanks. According to the "Separation Rules" given in the section on "Program Syntax", these blanks can be replaced by newlines, tabs, and comments; and, in some places, blanks can be inserted or omitted. Thus, for example, a SIGNAL statement can be written as:

```
SIGNAL
FIXEDOVERFLOW;
```

This statement was obtained by replacing the first blank as a newline and omitting the blank between the condition designator and the semicolon.

LIST OF ITEMS

In a syntax diagram, the characters ', ...' are not interpreted literally; instead, they indicate a possibly vacuous list of items separated from one another by commas. Similarly, the characters '...' indicate a list of items separated by blanks. The items in the list are specified by the construct that precedes the ', ...' or '...'.

As an example, consider the syntax diagram for the assignment statement, which is:

target, ... = e ;

This rule is a short way of specifying one of the following forms:

target = e ;

target , target = e ;

target, target, target = e ;

(etc.)

This notation does not imply that the targets must be identical; for example, a valid assignment statement is:

```
S3(2),X,ALPHA = M;
```

This statement has three different targets.

As a second example, consider the syntax diagram for the FREE statement, which is:

```
FREE  { ref1  IN(ref2) }  , ... ;
```

Here, each item in the list has the form:

```
ref1  IN(ref2)
```

This example shows that the construct that precedes the ', ...' can be a sequence of constructs enclosed in curly braces.

CHOICE OF ITEMS

In a syntax diagram, a vertical list of items enclosed in curly braces indicates a choice among those items. As an example, consider the syntax diagram for the GOTO statement:

```
{ GOTO  } ref ;
{ GO TO }
```

This rule is a short way of specifying the following forms:

```
GOTO  ref ;
GO TO  ref ;
```

OPTIONAL ITEMS

In a syntax diagram, a construct enclosed in square brackets is optional. As an example, consider the syntax diagram for the IF statement:

```
IF  e  THEN  ex1  [ELSE  ex2]
```

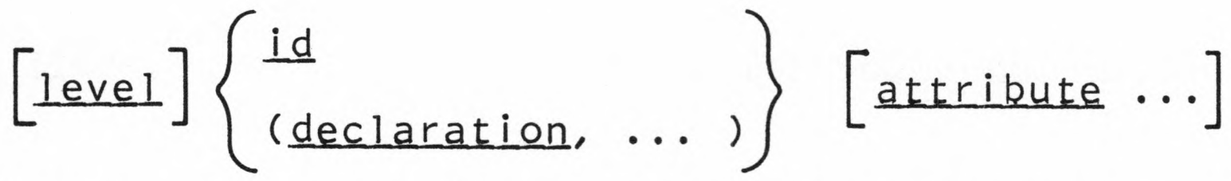
This rule is a short way of specifying the following forms:

```
IF  e  THEN  ex1  ELSE  ex2
IF  e  THEN  ex1
```

Not all optional features of statements are marked as such in the syntax diagrams; too many brackets obscure the syntax diagram. However, a construct that is optional is always mentioned in the supplementary rules that follow a syntax diagram. This treatment is accorded primarily to the PL/I constructs called options.

RECURSIVE DIAGRAMS

Occasionally, a diagram that defines a certain syntax variable also contains that syntax variable. As an example, consider the following definition for declaration:



In this diagram, the choice of the first alternative in the braces avoids recursion and produces something like:

```
ALPHA FIXED EXTERNAL
```

However, the choice of the second alternative introduces a parenthesized list of declarations; for example:

```
(SYSIN,SYSPRINT) FILE
```

or even:

```
(X, (Y,Z) FLOAT, BETA) CONTROLLED
```

The Parts of a Statement

A statement is composed of a prefix, followed by statement body, followed by a semicolon. A prefix is composed of a sequence of any number (possibly zero) of condition prefixes followed by a sequence of any number (possibly zero) of label prefixes. For some statements (DECLARE and DEFAULT), a condition prefix must not be used. For some statements (PROCEDURE, ENTRY, and FORMAT) a label prefix must be used.

A condition prefix has the form:

```
( id, ... )
```

where each id is an identifier indicating the enabling or disabling of a condition. A label prefix has the form:

```
[id   ( int )]  :
```

where id is an identifier and int is an optionally-signed decimal integer.

The parts of a statement are described fully in the section on "Program Syntax". The role of the condition prefix is defined in the section on "Condition Handling". The role of the label prefix is defined in the section on "Program Flow".

Specific Conventions

Every reference or expression mentioned in the statement descriptions must yield a scalar value unless an exception is explicitly noted.

The following syntactic variables are used throughout this appendix with the given meaning:

e, e1, e2, and so on, represents an arbitrary expression

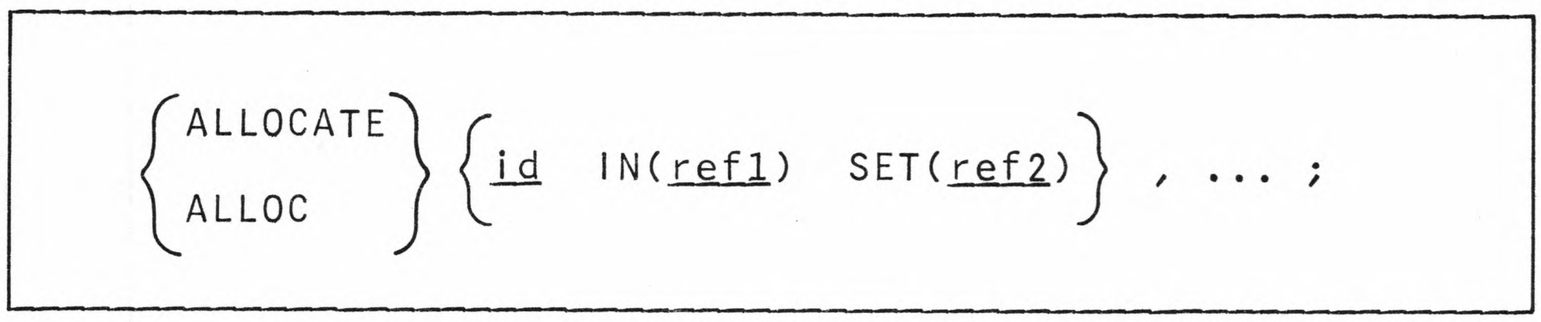
ref, ref1, ref2, and so on, represents an arbitrary reference to a constant, variable, or function

id represents an arbitrary identifier

target represents an arbitrary variable reference or pseudo-variable

Other syntactic variables are used for specific statements, and their meanings are given in the supplementary rules that follow the diagram in which they are used.

ALLOCATE



where id must be a CONTROLLED or BASED variable name, ref1 must yield an area variable, and ref2 must yield a locator variable. The options are selected as follows:

- If id is CONTROLLED, then both options must be omitted.
- Suppose id is BASED. If id is declared BASED(q), then the SET option can be omitted and SET(q) is assumed; otherwise, the SET option must be given.
- If id is a BASED variable intended for allocation in system storage rather than in an area variable, then the IN option must be omitted.
- Suppose id is a BASED variable intended for allocation in an area. If ref2 is declared OFFSET(a), then the IN option can be omitted and IN(a) is assumed; otherwise, the IN option must be given.

The options can be given in any order, but the order shown above is recommended.

The statement allocates storage. If id is CONTROLLED, storage is allocated in system storage and is stacked on any previous allocations of id. If id is BASED, storage is allocated either in system storage or the area given by the IN option; then the locator variable given in the SET option is set to designate the allocated storage.

ASSIGN

```
target, ...  =  e ;
```

The expression e is evaluated and its value is assigned to the storage specified by each target. The expression e can be any kind of PL/I expression, scalar or aggregate. A target can be a variable reference or a pseudo-variable. Each target must designate a storage unit that can accommodate the value of e after the value has been subjected to an allowed conversion of its storage type. See the section on "Value Assignment".

BEGIN

```
BEGIN ;
```

The statement is the first statement of a BEGIN block. A BEGIN block has two purposes: it groups together the statements contained in the block and it delimits the scope of the names declared in the block. See the sections on "Program Syntax" and "Program Flow".

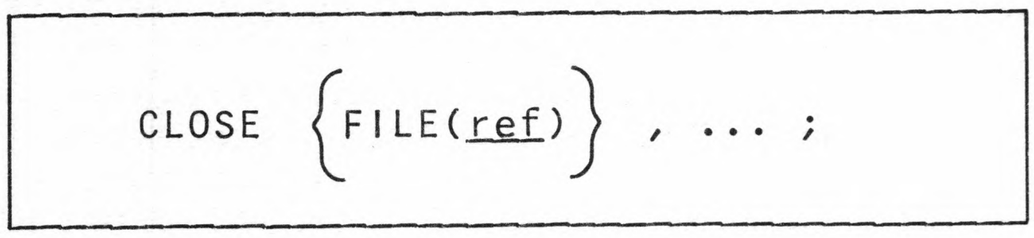
CALL

```
CALL  ref( e, ... ) ;
```

where, if there are no arguments, the parenthesized argument list, '(e, ...)', can be omitted or written as '()'.

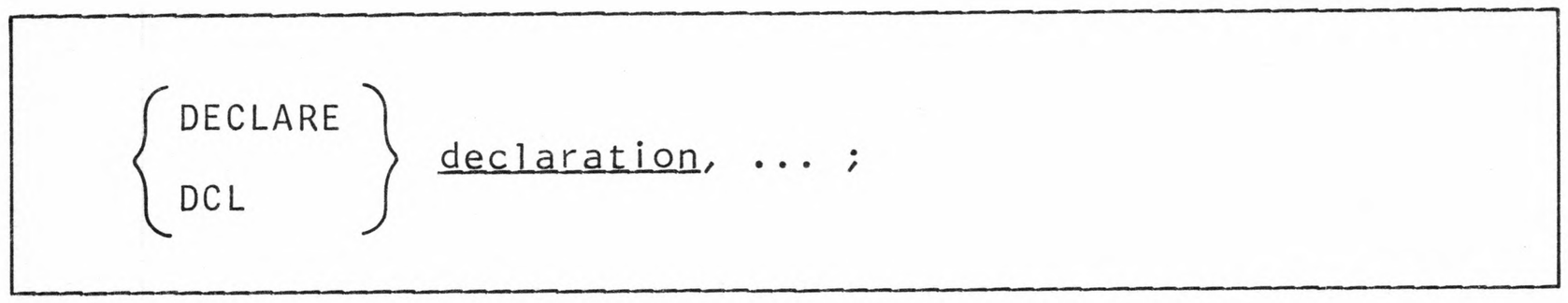
The statement invokes a procedure. The ref is evaluated to give an entry value. A procedure block is entered at the PROCEDURE statement or ENTRY statement designated by the value of ref and the procedure is executed for the given argument list. The argument list must contain one argument for each parameter in the statement at which the procedure is entered. The procedure must not return a value for an invocation by a CALL statement. See the section on "Procedure Invocation".

CLOSE



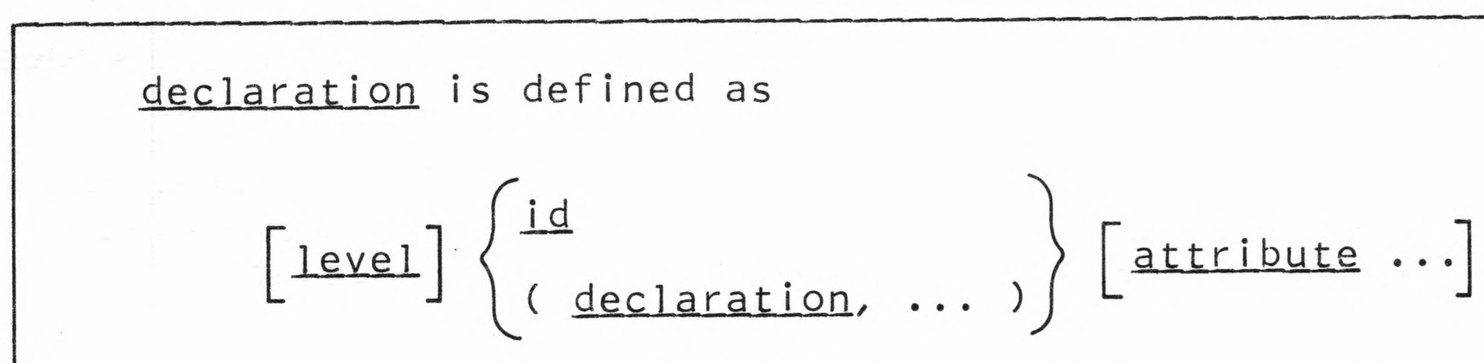
The statement closes one or more files. Each ref specifies a file value, and the corresponding file is closed if it is not already closed. See the sections on "Stream Input/Output" and "Record Input/Output".

DECLARE



where

declaration is defined as



and where level is an unsigned decimal integer. The statement must not be preceded by a condition prefix.

The statement declares one or more identifiers. An identifier is declared by associating with it a set of attributes and, in the case of a component of a structure, a level number. See the section on "Declarations".

DEFAULT

The DEFAULT statement allows a programmer to extend and modify the defaults that are built into PL/I for the declaration of identifiers; in addition, it allows the programmer to specify declarations that he wishes to have flagged as errors even though they are otherwise valid in PL/I. The DEFAULT statement is not recommended for use in an individual program; rather, its application lies in the establishment and maintenance of special standards for all programs written for a given project or at a given computing installation. The DEFAULT statement is not described here, but a full description appears in the PL/I Language Manual.

DELETE

```
DELETE  FILE(ref)  KEY(e) ;
```

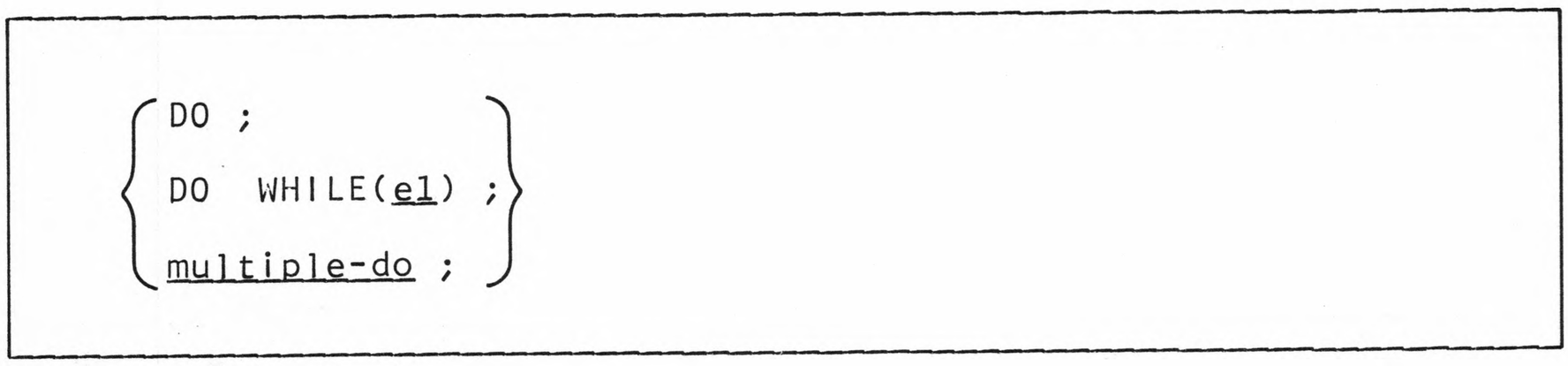
The options are selected as follows:

- The KEY option must be omitted if the file is not KEYED.
- The KEY option cannot be omitted if the file is DIRECT.

The options can be given in any order, but the order shown above is recommended.

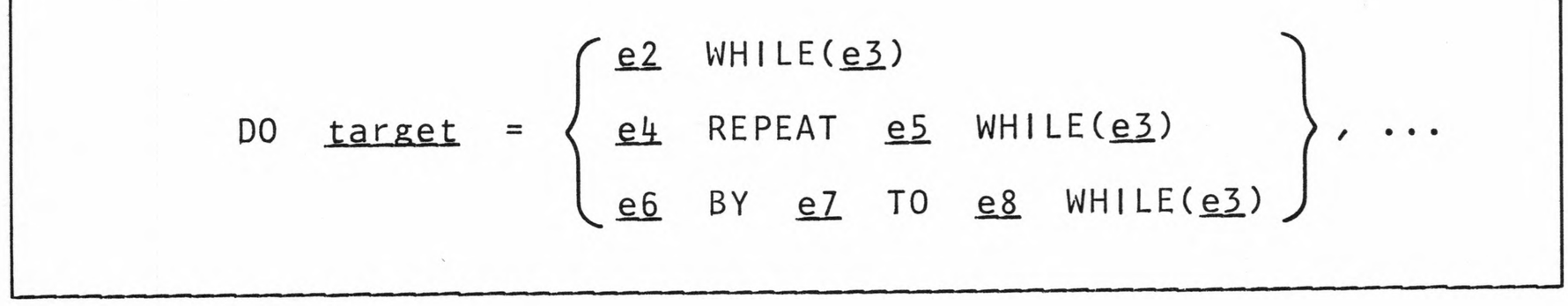
The statement deletes a record from a data set. The record is specified by the file value given by the FILE option and the character-string value given by the KEY option (if present). See the section on "Record Input/Output".

DO



where

multiple-do is defined as



The options and clauses are selected as follows:

- The option WHILE(e3) can be omitted. In that case, WHILE("1"B) is assumed.
- The BY e7 clause can be omitted. In this case, BY 1 is assumed.
- The TO e8 clause can be omitted, provided that the BY clause is not omitted. In this case, the end test associated with the TO clause is not performed.

The BY clause and the TO clause can be interchanged.

The first of the three forms, 'DO;', is the first statement of a noniterative group. The statements contained in the group are executed exactly once.

The second of the three forms, 'DO WHILE(e1);', is the first statement of an unindexed iterative group. Execution of the group begins with a while test. The expression e1 is evaluated and must yield a bit-string value. If all of the bits are zero bits, then execution is complete; otherwise, the statements in the group are executed and another execution of the group begins (with another while test).

The third of the three forms, DO target = ... , is the first statement of an indexed iterative group. The statement provides an index, specified by target, that is used to control the repeated execution of the statements in the group. As the second syntax diagram shows, there are three ways to control the index; their interpretation is not given here. See the section on "Program Flow".

END

```
END  [id  ;]
```

This statement is the last statement of a group, a BEGIN block, or a procedure. The optional identifier id is a closure label; its use is not recommended. See the sections on "Program Syntax", "Program Flow", and "Procedure Invocation".

ENTRY

```
ENTRY( id, ... ) [RETURNS( dec1 )] ;
```

where

- The statement must be immediately contained in a procedure.
- The statement must begin with at least one label prefix; however, no label prefix can have a subscript.
- If there are no parameters, the parameter list '(id, ...)' can either be omitted or written as '()'.
- Each identifier in the parameter list must be a level-one variable name declared in the immediately containing procedure.
- The number of parameters in the parameter list must equal the number of arguments in the argument list of the CALL statement or function reference that invoked this entry.
- The RETURNS attribute must be omitted or must appear depending on whether the entry is invoked by a CALL statement or a function reference.
- dec1 is the declaration of the value that is returned by the procedure when it is invoked at this entry.

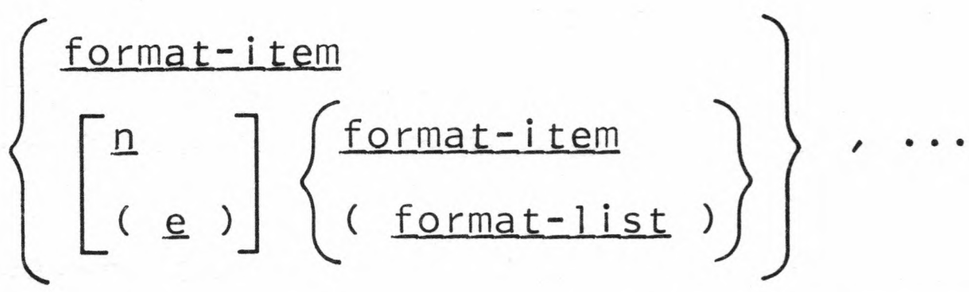
The statement provides an additional entry to a procedure. The entry can differ from other entries in its position within the procedure, in its parameter list, and in its RETURN attribute. Thus, one procedure can be invoked in several different ways if ENTRY statements appear in the procedure.

FORMAT

```
FORMAT  ( format-list )  ;
```

where the statement must be preceded by at least one label-prefix and

format-list is defined as



where n is an unsigned integer and the format-items are as follows:

```
A(w)              -- transmit a character-string value representation
B(w)              -- transmit a bit-string value representation
F(w,fw,dm)        -- transmit a fixed-point value representation
E(w,fw,ms)        -- transmit a floating-point value representation
C(part1,part2)    -- transmit a complex value representation
P"x"              -- transmit a pictured value representation
X(e)              -- skip e character positions
COLUMN(e)         -- skip to column e of a line
SKIP(e)           -- skip e lines
LINE(e)           -- skip to line e of a page
PAGE              -- skip to the next page
R(ref)            -- invoke another format-list for the next format-item
```

In these format items, w (width) is the number of characters in an input or output field, fw (fraction width) is the number of fractional digits in the value representation, dm (decimal multiplier) is a power of ten, ms (mantissa significance) is the number of digits in the mantissa of a floating-point value representation, part1 and part2 can each be any F, E, or P format item, x must be a PL/I picture, e is an arithmetic expression, and ref is a format-valued reference. Some of the arguments of format-items can be omitted and default values are then assumed.

The statement supplies a format-list for use in an edit-directed stream input/output statement. The format-list provides a format item for each value transmitted and also provides format items to skip spaces, lines, and pages between value representations. See the section on "Stream Input/Output".

```
FREE { ref1  IN(ref2) }  , ... ;
```

The option is selected as follows:

- The IN option must be omitted if the storage being freed is a controlled variable.
- The IN option must be omitted if the storage being freed is a based variable that is allocated in system storage.
- The IN option can be omitted in any case in GCOS PL/I; but Standard PL/I requires the option for a based variable allocated in an area.

The statement frees the controlled or based variable designated by ref1. See the section on "Storage Management".

GET

```
GET { FILE(ref1)  COPY(ref2)  SKIP(e1) }  input-option ;
    { STRING(e2)  COPY(ref2)            }
```

where

input-option is defined as

```
{ DATA( data-ref, ... )                                  }
{ LIST( get-item, ... )                                  }
{ EDIT { ( get-item, ... ) ( format-list ) } ...         }
```

get-item is defined as

```
{ target                                  }
{ ( get-item, ...  multiple-do )          }
```

and where

- The data-ref is a simple or unsubscripted structure-qualified variable reference.
- The format-list is defined under the FORMAT statement in this appendix.
- The multiple-do is defined under the DO statement in this appendix.

The options are selected as follows:

- The FILE and STRING options can be omitted. In this case, FILE(SYSIN) is assumed.
- The COPY option can be omitted. In this case, no copy of the input is made.
- The parenthesized argument '(ref2)' in the COPY option can be omitted. In this case, COPY(SYSPRINT) is assumed.
- Either the SKIP option or the input option (but not both) can be omitted. In this case, the corresponding skip or input is not performed.

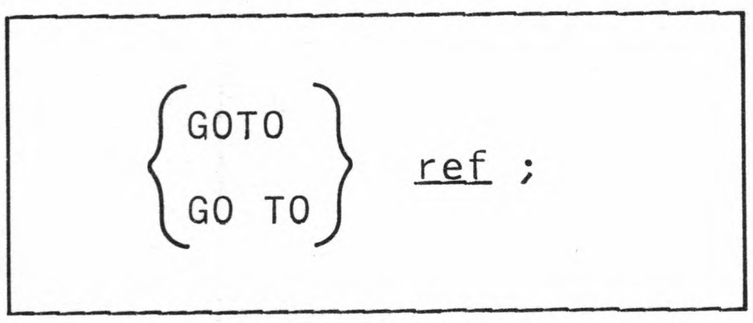
- The parenthesized argument '(e1)' in the SKIP option can be omitted. In this case, SKIP(1) is assumed.

- The list of input items '(data-ref, ...)' can be omitted from the DATA input option. In this case, a list of input items including every level-one variable accessible from the GET statement is assumed. This variation is expensive, and should not be used without due consideration of the cost.

The options can be given in any order, but the order shown above is recommended.

The statement reads value representations from a stream data set and assigns their values to program variables. See the section on "Stream Input/Output".

GOTO



The statement transfers control to the statement specified by the value of ref. See the section on "Program Flow".

IF

```
IF  e  THEN  ex1  [ELSE  ex2]
```

where ex1 and ex2 are executable units. An executable unit is defined as

a group (a DO statement, other statements, and an END)
a begin-block (a BEGIN statement, other statements, and an END)
an independent statement (any statement which acts alone).

An independent statement is any statement except the following:

declarative statements: DECLARE and DEFAULT
dependent statements: FORMAT,DO,BEGIN,PROCEDURE,ENTRY,END.

The ELSE ex2 clause can be omitted, giving a statement of the form

IF e THEN ex1

The statement evaluates e to produce a bit-string value. If the bit-string value contains a 1 anywhere (and therefore represents "true"), then ex1 is executed and ex2 is skipped. If the bit-string contains only 0 bits, then ex1 is skipped and ex2 is executed. If the ELSE clause is omitted, then no action is performed when the bit-string contains only 0 bits.

LOCATE

```
LOCATE  id  SET(ref1)  FILE(ref2)  KEYFROM(e) ;
```

The options can be omitted as follows:

- The SET option can be omitted provided that id is declared (elsewhere) as BASED(q), where q is a locator qualifier. In this case, SET(q) is assumed.
- The FILE option cannot be omitted.
- The KEYFROM option must be omitted if the file designated by ref1 does not have the KEYED attribute.

The options can be given in any order, but the order shown above is recommended.

The statement allocates storage for the based variable id and sets the locator variable specified by ref1. The allocated storage serves as a buffer, and values can be assigned to it by subsequent statements. Then, when the next output statement is encountered (or the file is closed), the contents of the buffer are output. Output is directed to the file designated by ref2 using the key given by e (if the KEYFROM option occurs). See the section on "Record Input/Output".

NULL

```
;
```

The statement does nothing. It can be used where an executable-unit is called for, as in the IF and ON statements. See the sections on "Program Flow" and "Condition Handling".

ON

```
ON cr [SNAP] ON unit ;
```

where cr is a condition reference and ON unit is

a BEGIN block
a statement
the keyword SYSTEM

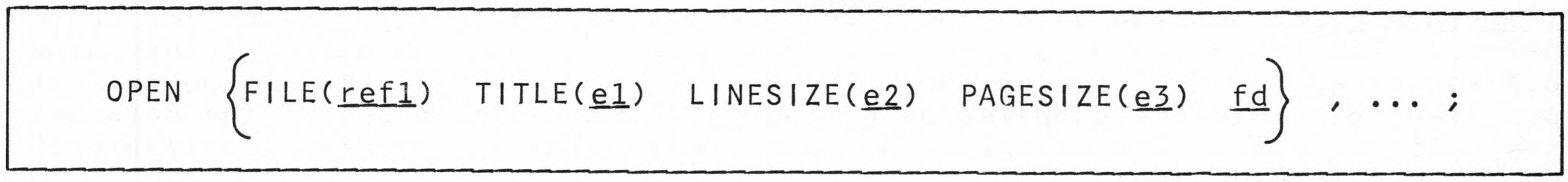
If the ON unit is a BEGIN block, then a RETURN statement can appear only if it is within a procedure within the BEGIN block. If the ON unit is a statement, then it must not be any of the following:

declarative statements:	DECLARE and DEFAULT.
dependent statements:	FORMAT,DO,BEGIN,PROCEDURE,ENTRY,END.
other excluded statements:	ON,REVERT,IF,RETURN.

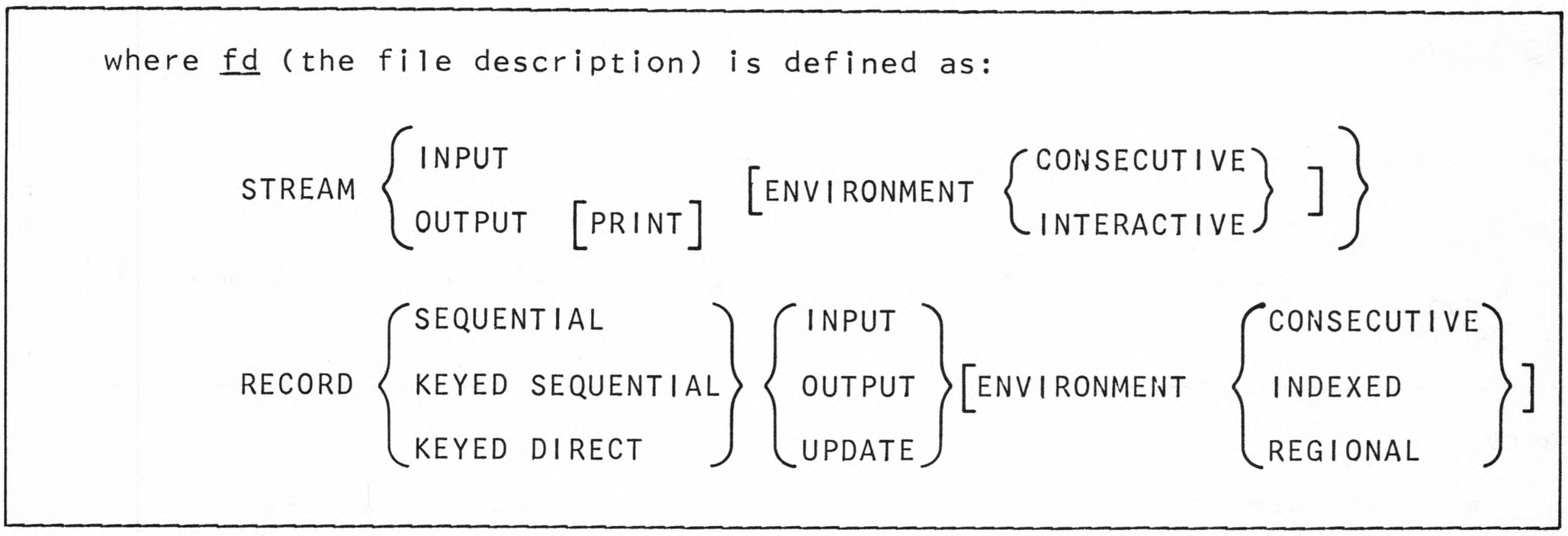
If an ON statement begins with a condition prefix, that prefix applies only to the evaluation of the condition name. If the ON unit is a BEGIN block or a statement, it can have its own condition prefixes, but it cannot have a label prefix.

The statement establishes the ON unit as the action which will be taken when the condition, cn is signalled. If SNAP appears, the active block names and their arguments are printed. If SYSTEM is used as the ON unit, the PL/I default ON unit for the condition is invoked.

OPEN



where



The options and attributes are selected as follows:

- The FILE option cannot be omitted.
- The TITLE option can be omitted. In this case, the file code is determined from the filename.
- The LINESIZE and PAGESIZE options must be omitted unless the file description includes OUTPUT PRINT.
- The LINESIZE option can be omitted when the file description includes OUTPUT PRINT. In this case, LINESIZE(132) is assumed for most devices (including the printer).
- The PAGESIZE option can be omitted when the file description includes OUTPUT PRINT. In this case, PAGESIZE(55) is assumed for most devices (including the printer).
- The STREAM and RECORD attributes can be omitted because they can be deduced from other attributes given in the file description.
- The OUTPUT attribute can be omitted if the file description contains the PRINT attribute.
- The ENVIRONMENT attribute can be abbreviated to ENV.
- The KEYED attribute can be omitted if the file description includes the DIRECT attribute.

The statement supplies information to control the transmission of data between a data set and program storage. The information is stored in a file-state block designated by the file value supplied by ref1 in the FILE option. The data set to be used is designated by the character-string value supplied by e1 in the TITLE option. The format of a printed page of output is specified by integers supplied by e2 and e3. The intended use of the data set is specified by the file description. See the sections on "Stream Input/Output" and "Record Input/Output".

PROCEDURE

```
{PROCEDURE}
{PROC     } ( id, ... ) [RETURNS( dec1 )][RECURSIVE] [OPTIONS( MAIN )];
```

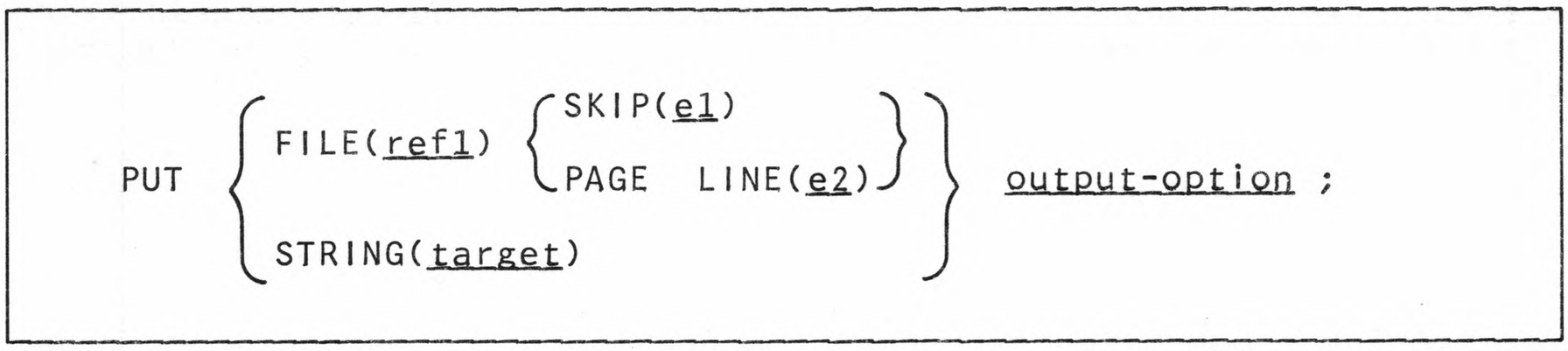
where

- The statement is the first statement of a procedure block.
- The statement must begin with at least one label prefix; however, no label prefix can have a subscript.
- If there are no parameters, the parameter list '(id, ...)' can either be omitted or written as '()'.
- Each identifier in the parameter list must be a level-one variable declared immediately within the procedure.
- The number of parameters in the parameter list must equal the number of arguments in the argument list of the CALL statement or function reference that invoked this entry.
- The RETURNS attribute must be omitted or must appear according to how the procedure is invoked by a CALL statement or a function reference.
- dec1 is the declaration of the value which is returned by the procedure when it is invoked at this entry.
- The RECURSIVE keyword must be used in Standard PL/I if the procedure is recursive. This keyword is ignored by GCOS PL/I, which assumes that all procedures are recursive.
- The keyword MAIN identifies the procedure to which GCOS passes control.

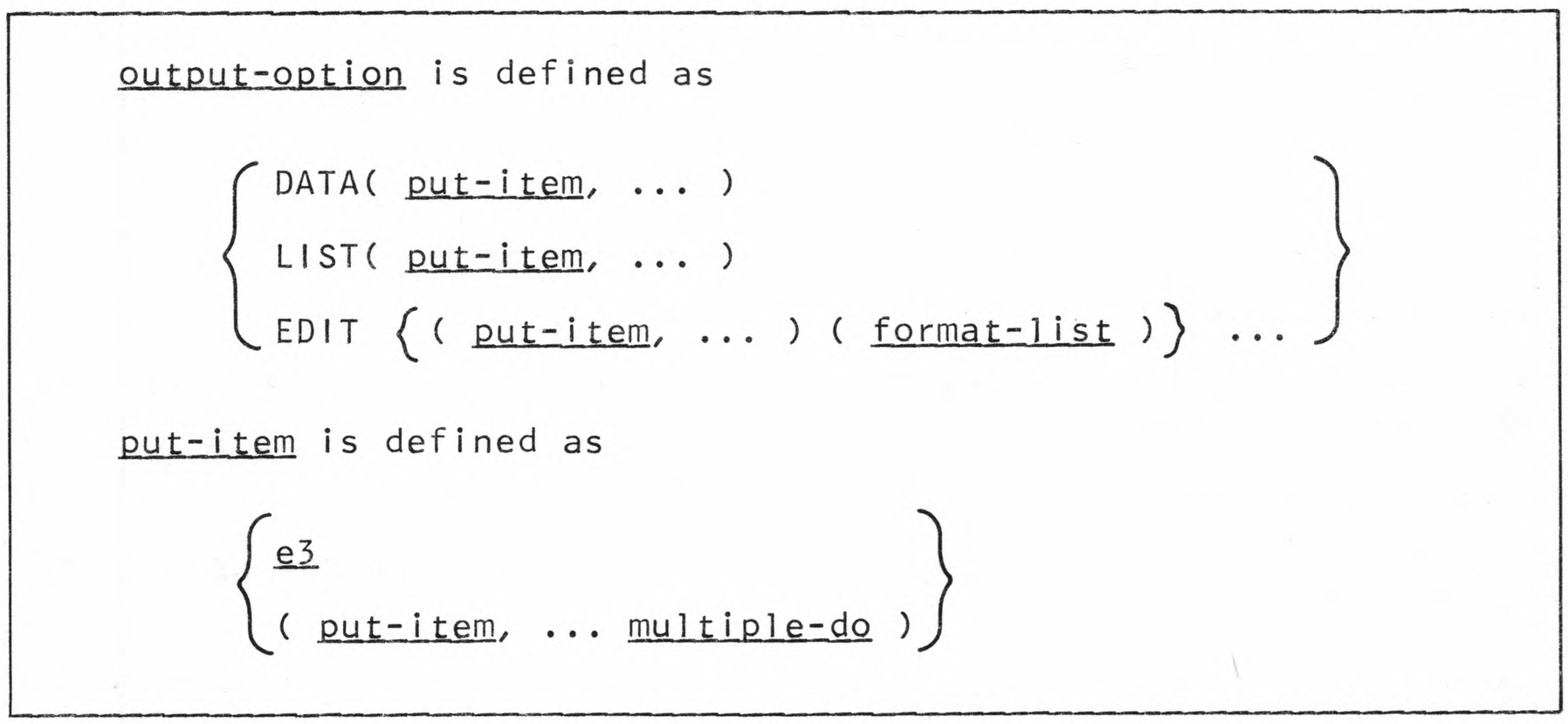
The order in which the RETURNS and OPTIONS attributes and RECURSIVE keyword are given is optional, but the order given above is recommended.

The statement provides the principal entry to a procedure. One or more ENTRY statements can be used to provide additional entries to the same procedure.

PUT



where



and where

- The expression, e3, in a put-item for a DATA option is restricted to a variable reference.
- The format-list is defined under the FORMAT statement in this appendix.
- The multiple-do is defined under the DO statement in this appendix.

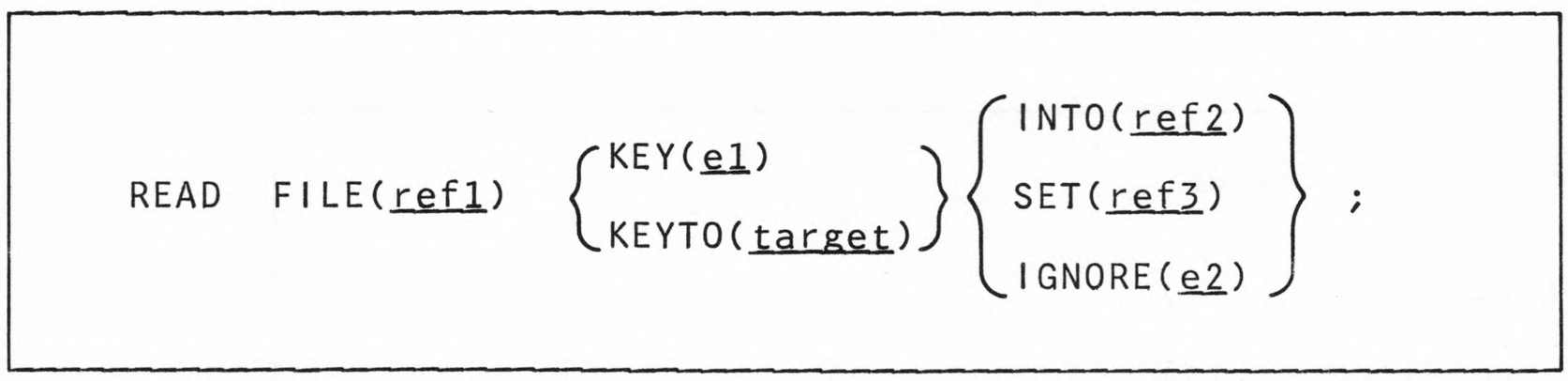
The options are selected as follows:

- The FILE and STRING options can be omitted. In this case, FILE (SYSPRINT) is assumed.
- The SKIP option, PAGE option, LINE option, and output-option can be omitted in any way provided one of them remains. In these cases, the corresponding output operation will not occur.
- The parenthesized argument '(e1)' in the SKIP option can be omitted. In this case, SKIP(1) is assumed.
- The list of output items '(put-item, ...)' can be omitted from the DATA output option. In this case, a list of output items including every level-one variable accessible from the PUT statement is assumed.

The options can be given in any order, but the order shown in the syntax diagram is recommended.

The statement takes values from program storage (or computes values) and writes their representations into a stream data set. See the section on "Stream Input/Output".

READ



The options are selected as follows:

- The KEY option must be omitted if the file is not KEYED or if the IGNORE option is used.
- The KEY option cannot be omitted if the file is DIRECT.
- The KEYTO option must be omitted if the file is not KEYED SEQUENTIAL or if the IGNORE option is used.

The options can be given in any order, but the order shown above is recommended.

The statement reads a record from a data set or skips records in a sequential data set. The data set is specified by the file value given in the FILE option. The KEY option gives a character-string value used to locate a record. The KEYTO option gives a storage unit suitable for a character-string value representing a key. The INTO option provides storage for an input value. The SET option provides storage for a pointer to a system buffer into which (if this option is used) the record will be read. The IGNORE option gives the number of records to be skipped as an integer value. See the section on "Record Input/Output".

RETURN

```
RETURN  [( e )] ;
```

The parenthesized expression must appear if the statement returns to a function reference, and must be omitted if the statement returns to a procedure call.

The statement terminates execution of a procedure and returns control to the function reference or CALL statement that invoked it. See the section on "Procedure Invocation".

REVERT

```
REVERT  cn, ... ;
```

where cn must be a condition name.

The statement reverts each condition name provided it was established by an ON statement in the current block. See the section on "Condition Handling".

REWRITE

```
REWRITE  FILE(ref1)  KEY(e1)  FROM(ref2) ;
```

The options are selected as follows:

- The KEY option must be omitted if the file is not KEYED.
- The KEY option cannot be omitted if the file is DIRECT.

The options can be given in any order, but the order shown above is recommended.

The statement replaces a record in a data set. The record to be replaced is specified by the file value given by the FILE option and the character-string value given by the KEY option. The new value for the record is supplied by the reference in the FROM option. See the section on "Record Input/Output".

SIGNAL

```
SIGNAL  cr  ;
```

where cr must be a condition reference.

The statement signals the condition designated by cr. See the section on "Condition Handling".

WRITE

```
WRITE  FILE(ref1)  KEYFROM(e)  FROM(ref2) ;
```

The options are selected as follows:

- The KEYFROM option must be omitted if the file is not KEYED.
- The KEYFROM option cannot be omitted if the file is DIRECT.

The options can be given in any order, but the order shown above is recommended.

The statement adds a record to a data set. The destination of the record is specified by the file value given by the FILE option and the character-string value given by the KEYFROM option (if present). The value for the record is supplied by the reference in the FROM option. See the section on "Record Input/Output".

INDEX

9 indicator
  non-numeric 3-43
  numeric 3-30

$ indicator
  drifting 3-37
  leftmost 3-34

% INCLUDE macros 5-7

' 9-41

' operator 9-41

'' operator 9-32

'&' operator 9-41

'' extent 12-7

'' operator 9-7

'*' operator 9-9

'+' operator
  infix 9-6

'+' operator prefix 9-6

'-' operator
  infix 9-6

'-' operator prefix 9-6

'/' in comment 5-10

'/' operator 9-8

'<' operator
  for arithmetic values 9-10
  for string values 9-38

'<=' operator
  for arithmetic values 9-10
  for string values 9-38

'=' operator
  for address values 9-48
  for arithmetic values 9-10
  for string values 9-38

'>' operator
  for arithmetic values 9-10
  for string values 9-38

'>=' operator
  for arithmetic values 9-10
  for string values 9-38

'¬' operator 9-41

'¬<' operator
  for arithmetic values 9-10
  for string values 9-38

'¬=' operator
  for address values 9-48
  for arithmetic values 9-10
  for string values 9-38

'¬>' operator
  for arithmetic values 9-10
  for string values 9-38

* indicator 3-35

+ indicator
  drifting 3-36
  leftmost 3-33

, indicator 3-38

- indicator
  drifting 3-36
  leftmost 3-33

. indicator 3-38

/ indicator 3-38

A format item 14-28

A indicator 3-43

abbreviations and defaults
  alignment attributes 3-63
  AREA attribute 3-48
  arithmetic attributes 3-8
  definition of default 3-9
  DIMENSION attribute 3-55
  files 14-10
  guidelines for use 3-8
  line length 14-9
  management class attributes 7-11
  ordinary string attributes 3-17

abbreviations and defaults (cont)
page length 14-9
picture attributes 3-23
shortened variable references 8-27

ABS function 9-16

activation indexes 12-37

activation internal regions 7-4

activation regions
for procedure activation 12-10
in recursion 12-29

activation variable reference 12-40

ADD function 9-10

ADDR function 7-25, 9-49

ADDREL function 9-51

addresses
assignment of 10-2
attributes 3-45
implicit locator targets 4-6
operations 9-48
values 2-3

AFTER function 9-35

aggregate
assignment of 10-3
expressions 8-5

aggregates
aggregate type
role of 3-2
3-49
guidelines for use 3-60
operations on 9-2
values 2-4

algebraic comparison operators 9-10

ALIGNED attribute 3-62

alignment types
attributes 3-61
role of 3-2

ALLOCATE statement
for BASED variables 7-22
for CONTROLLED variables 7-18
syntax diagram A-6

ALLOCATION function 9-71

allocation of storage 7-1

ambiguous declaration 6-19

and operator 9-41

applicability of declarations 6-17

approximation of arithmetic values
for conversion 4-8

AREA
assignment of 10-3, 10-7
attribute 3-47
condition 7-40
initialization of 7-6
operations 9-48
size 3-47
storage 3-46
values 2-4

AREA function 9-52

argument evaluation
in operations 9-2

arguments
by-reference 12-3
by-value 12-3
classification of 12-3
connected and unconnected 12-5
guidelines for 12-8
interpretation of 12-6, 12-10
passing of 12-2

arithmetic
assignment 10-1
constant literals 8-31
constants
as lexemes 5-5
data types 3-5
guidelines for data 3-14
operations 9-3
storage 3-4
values 2-1

arithmetic decimal-point indicator
3-39

array
values 2-4

arrays
DIMENSION attribute 3-54
operations 9-52
storage 3-57
storage types 3-56
3-53

ASCII characters
new line 14-2
new page 14-2
2-2, 3-16

assignment statements
AREA assignments 10-7
form of 10-4
guidelines for 10-8
interpretation of 10-5
order of interpretation 10-7
overlapping string targets 10-5
pseudo-variables 10-9
restrictions 10-5
special string target 10-6
syntax diagram A-7
10-1

associated storage types 8-22

asterisk extent 12-7

ATAN function
  Cartesian 9-26
  ordinary 9-25

ATAND function
  Cartesian 9-27
  ordinary 9-26

ATANH function 9-29

attributes
  classification of 6-24
  complete sets
    for built-in function names 6-23
    for condition names 6-23
    for constant literals 8-36
    for constant names 6-22, 8-41
    for generic names 6-23
    for variable names 6-21
    6-20
  6-20

AUTOMATIC variables
  attribute 7-10
  guidelines for 7-32
  7-15

B format item 14-28

B indicator 3-38

base attribute
  guidelines for choice of 3-15

base attributes 3-6

base variable 7-25

based input 15-15

based output .15-19

BASED variables
  attribute 7-10
  guidelines for 7-34
  7-20

BEFORE function 9-35

BEGIN block
  as executable unit 11-4, 11-19
  as program structure 5-18
  in general recursion 12-42

BEGIN statement
  syntax diagram A-7

BINARY
  attribute 3-6
  function 9-61

BIT
  attribute 3-16
  function 9-64

BIT (cont)
  operations 9-41
  3-16

BIT values 2-3

block
  activation of 11-19
  as program structure 5-18
  exit from 11-16

body
  of DO group 11-9

body of statement 5-14

books on PL/I 1-12

BOOL function 9-42

Boolean values 2-3

bounds of array 3-54

box of storage unit 3-2

break character 5-3

built-in function names
  attributes for 6-23
  8-46

built-in function references 8-45

by-reference arguments 12-3

by-value arguments 12-3

C format items 14-33

CALL statement
  execution of 12-9
  syntax diagram A-7
  12-9

capacity of storage 7-38

CEIL function 9-12

chained recursion 12-29

CHARACTER
  attribute 3-16
  function 9-63
  operations 9-45
  values 2-2

characters
  classification of 5-2
  2-2, 3-16

classification
  of arguments 12-3
  of attributes 6-24
  of characters 5-2
  of conditions 13-2
  of lexemes 5-12
  of operators 5-6

classification (cont)
of picture indicators 3-21
of pictures 3-22
of statements 5-16
of values 2-5

clause
declaration 6-5

clauses
general form 5-14

CLOSE statement
for record files 15-8
for streams 14-9
syntax diagram A-8

COLLATE function 9-45

collating sequence 9-39, 9-46

COLUMN format item 14-39

column position
initialization of 14-8
14-5

combined declarations 6-6

comments
as lexemes 5-10

common data attributes 9-5

comparison operations
for value addresses 9-48
9-38

comparison operators
for arithmetic values 9-10

complete attribute sets 6-20

COMPLEX
attribute 3-5, 3-23
function 9-18, 9-59
values 2-1

complex arithmetic function 9-17

complex format items 14-33

components of aggregate 3-49

compound list items
array variable names 14-21
iterated list 14-22
structure variable names 14-22
14-21

computational conditions 13-2

computational values 2-5

concatenate operator 9-32

CONDITION
attribute 13-7

condition names
attributes for 6-23

condition prefix
purpose of 5-13
scope of 13-8

conditions
built-in functions 13-15
classification of 13-2
declaration of 13-7
default enabling 13-9
fatal 13-14
for conversion 4-20
for storage management 7-39
for stream input/output 14-44
guidelines for 13-18
language-defined
computational 13-2
input/output 13-5
storage 13-4
termination 13-5
13-2
occurrence of 13-12
principal features 13-1
programmer-defined 13-6
references 13-6
signalling of 13-12
13-1

conditon prefix 13-8

CONJG function 9-19

consequences of IF statement 11-4

constant literals
attribures for 8-36
for arithmetic values 8-31
for string values 8-34
8-31

constant names
attributes for 6-22

constant reference
as destination of GOTO 11-16

constant references
to external entries 8-39
to files 8-40
to record files 15-4
to statement addresses 8-37
to stream files 14-6
8-36

CONSTANT storage
attribute 7-10
7-13

containing references 8-16

containment
definition of 6-3

contents
of storage unit 3-2

contextual declaration 6-10

control characters for streams 14-2

control format items 14-38

control list in DO statement 11-10

CONTROLLED variables
  attribute 7-10
  guidelines for 7-34
  stacking of 7-18
  7-18

CONVERSION condition
  for arithmetic target 4-7
  for bit-string target 4-16
  14-45, 4-21

conversion of values
   guidelines for functions 9-68
  conditions for 4-20
  contexts
    arguments and results 4-3
    assignment statements 4-2
    assignment-like contexts 4-3
    bit-string targets 4-6
    built-ins and expressions 4-4
    character-string targets 4-5
    integer targets 4-5
    locator targets 4-6
    4-1
  operations for 9-55
  targets 4-2
  to aggregate targets 4-18
  to arithmetic targets 4-7, 9-58
  to bit-string targets 4-16, 9-63
  to character-string targets 4-10,
    9-63
  to locator targets 4-18, 9-65
  4-1

CONVERT function 9-56

COPY function 9-32

COPY option
  in GET statement 14-11

COS function 9-24

COSD function 9-25

COSH function 9-28

CR indicator 3-33

credit indicator 3-33

cross-section of array 8-12

current record indicator
  initialization of 15-6
  15-2, 15-4

dangling ELSE clause 11-7

data
  description of 1-2, 3-1
  storage 3-1
  values 2-1

data address values 2-5

data format items 14-27

data frame
  of storage unit 3-10

data sets
  record 15-1
  stream 14-1

data types
  addresses 3-45
  areas 3-47
  arithmetic 3-5
  complete attribute sets 6-21
  conversion of 4-1
  ordinary strings 3-16
  pictured strings 3-20
  role of 3-2

data-directed input/output 14-14

DATE function 9-70

DB indicator 3-33

debit indicator 3-33

DECAT function 9-36

DECIMAL
  attribute 3-6
  function 9-62

decimal-point indicator 3-32

declaration
  abbreviations and defaults 6-6
  ambiguous 6-19
  applicability 6-17
  clause of 6-5
  combined declarations 6-6
  contextual 6-10
  establishment of 6-1
  factored declaration 6-6
  implicit 6-10
  of built-in function names 6-23
  of condition names 13-7, 6-23
  of constant names 6-22
  of entry names 6-8
  of format names 6-9
  of generic names 6-23
  of label names 6-8
  of variable names 6-21
  resolution of names 6-15
  short forms of 6-6
  structure declarations 6-5
  6-1

declarations
  simple declarations 6-4

DECLARE statement
  sequential execution of 11-2
  syntax diagram A-8
  6-4

DECLARE STATEMENT
  guidelines for 6-7

DEFAULT statement
  sequential execution of 11-2
  syntax diagram A-9
  6-13

DEFINED variables
  attribute 7-10
  guidelines for 7-36
  7-29

DELETE statement
  for SEQUENTIAL input/output 15-13
  syntax diagram A-9

DELETE statement for KEYED
  input/output 15-10

descriptors
  for returned result 12-19

designator of storage unit 3-2

destination of GOTO statement 11-18

digit indicator 3-22

digits 5-2

DIMENSION attribute 3-54

DIMENSION function 9-53

dimensionality of array 3-54

DIRECT attribute 15-7

direct recursion 12-29

DIVIDE function 9-10

division operator 9-8

DO control
  FORTRAN 11-13
  index of 11-15
  REPEAT 11-12
  single value 11-11

DO group
  as consequence of IF 11-6
  as executable unit 11-4
  as program structure 5-17
  body of 11-9
  index of 11-15
  iterative with index 11-10
  iterative without index 11-9
  non-iterative 11-16
  11-8

DO statement
  syntax diagram A-10

dollar indicator 3-34

DOT function 9-55

double precision 3-7

drifting-dollar indicator 3-37

drifting-sign indicator 3-36

e (mathematical quantity) 9-21

E format item 14-31

E indicator 3-41

edit-directed input/output 14-25

elementary arithmetic operations 9-5

elements of arrays 3-53

ELSE clause 11-3, 11-7

EMPTY function 9-52

enabling conditions
  for debugging 13-18

END statement
  as procedure exit 12-11
  syntax diagram A-11
  12-22

end-around repetitions 14-41

ENDFILE condition
  for stream input/output 14-45

ENDPAGE condition 14-46

ENTRY
  attribute 3-45
  constant names
    declaration of 8-39
    8-37

entry declaration 12-23

entry names
  generic 12-28

entry points
  in a non-PL/I procedure 12-24
  in another external procedure 12-23
  in same external procedure 12-23

entry reference
  constant 12-23
  interpretation of 12-10
  12-23

entry references
  function 12-27
  variable 12-25

ENTRY statement
syntax diagram A-11
12-21

ENVIRONMENT attribute 15-7

ENVIRONMENT attribute interactive
streams
description of 14-9

equivalenced BASED variables 7-25

ERF function 9-30

ERFC function 9-30

ERROR condition 13-21

error function 9-30

established On unit 13-14

establishing ON units 13-10

establishment of declarations 6-1

executable unit 11-4

exit from block 11-16

EXP function 9-21

exponent
of FLOAT value 3-6

exponentiation operator 9-9

expressions
aggregate 8-5
infix 8-53
nested 8-2
operator expressions 8-52
parenthesized 8-3
prefix 8-53
priority of operators 8-54
8-1

extent functions 9-52

extents
evaluation of 7-5
for areas 3-47
for arrays 3-55
for AUTOMATIC variables 7-16
for BASED variables 7-23
for CONTROLLED variables 7-19
for DEFINED variables 7-29
for PARAMETER variables 7-20
for strings 3-16
or STATIC variables 7-17

external ENTRY constant names
declaration of 8-39

external procedures 5-20

external regions 7-4

EXTERNAL scope
attribute 7-10
7-14

F format item 14-29

F indicator
fixed-point 3-40
floating-point 3-42

factored declarations 6-6

false value 2-3

fatal conditions 13-14

field in stream 14-27

FILE
attribute 3-45
constant references 8-40
references
in condition references 13-6
to record files 15-4
values 2-4

file code 14-5, 15-4

file descriptions
for record files 15-7
for streams 14-9

file name 14-5, 15-4

FILE option
in CLOSE statement 14-9, 15-8
in DELETE statement 15-10, 15-13
in GET statement 14-11
in LOCATE statement 15-20
in OPEN statement 14-8, 15-6
in PUT statement 14-12
in READ statement 15-10, 15-12,
15-15
in REWRITE statement 15-11, 15-13
in WRITE statement 15-12, 15-8
WRITE statement 15-8

file-state blocks
for record files 15-3
for streams 14-5

files
references
for stream input/output 14-6

filler zeros 3-8

FINISH condition 13-21

FIXED
attribute 3-5
function 9-59

fixed-point format items 14-29

fixed-point PICTURE attributes 3-22

FIXEDOVERFLOW condition
for arithmetic target 4-8
for conversion 4-20

FLOAT
attribute 3-5
function 9-60

floating-point format items 14-31

floating-point indicator 3-41

floating-point PICTURE attributes
3-22

floating-point values 3-6

FLOOR function 9-12

flow of control
guidelines for 11-19
summary of 1-4
11-1

FORMAT
attribute 3-45

FORMAT constant names 8-37

format lists 14-41

FORMAT statement
in general recursion 12-42
sequential execution of 11-2
syntax diagram A-12
14-13

FORTRAN control for DO 11-13

FREE statement
for BASED variables 7-22
for CONTROLLED variables 7-18
syntax diagram A-13

FROM option
in REWRITE statement 15-11, 15-13
in WRITE statement 15-8, 15-12
omission of 15-19

fully-qualified reference 6-17

function references
interpretation of 12-15
result 12-16
12-14

GCOS files 15-2

GCOS PL/I User's Guide 1-14

general
recursion 12-36

generic entry names 12-28

generic names
attributes for 6-23

GET statement
syntax diagram A-14
14-11

GOTO statement
as procedure exit 12-11, 12-15
destination of 11-16
in general recursion 12-44
local attribute for 11-17
syntax diagram A-15
11-16

guidelines
for %INCLUDE macros 5-9
for abbreviations and defaults 3-8
for aggregates 3-60
for arguments 12-8
for arithmetic constants 8-34
for arithmetic data 3-14
for assignment statements 10-8
for built-in functions 8-50
for conditions
for conversion 4-22
for contextual declarations 6-11
for conversion functions 9-68
for data-directed input/output
14-19
for DECLARE statements 6-7
for DEFAULT statements 6-15
for edit-directed input/output
14-42
for flow af control 11-19
for identifiers 5-4
for implicity declarations 6-11
for LIKE attribute 6-13
for list-directed input/output
14-24
for pictured strings 3-44
for programmed functions 8-50
for programs 5-20
for storage class 7-32
for string data 3-19
for structures 3-50
for studying PL/I 1-12
for subscript list deletion 8-28
for variable references 8-31

guidelines for
scope attributes 7-14
condition handling 13-18

HBOUND function 9-53

HIGH function 9-47

hyperbolic functions 9-28

identifiers
as lexemes 5-3
guidelines for 5-4
maximum length of 5-3

IF statement
consequences of 11-4
dangling ELSE in 11-7
syntax diagram A-16
test in 11-4
within IF statement 11-6

IF statement (cont)
11-3

IMAG
function 9-18
pseudo-variable 10-9

imaginary outer block 6-4

imaginary part 10-9, 3-5

immediate containment
definition of 6-3

implicit declaration 6-10

implicit targets 4-4

IN option 7-22

INCLUDE macros 5-7

inclusive-or operator 9-41

increment of DO group 11-13

independent statements
as executable unit 11-5

INDEX function 9-34

index of DO group 11-10

infix operators 8-53

infix sign operators 9-6

INITIAL attribute
evaluation of 7-6
7-10, 7-36

initialization of DO group 11-13

INPUT attribute
for record files 15-7
for streams 14-9

input/output
conditons 13-5
summary of 1-5

insertion-character indicator 3-38

integers
implicit integer targets 4-5

internal regions 7-4

INTERNAL scope
attribute 7-10
7-13

INTO option
in READ statement 15-10, 15-12

isub DEFINED variables 7-31

isubs
as lexemes 5-7
restriction of 12-3

iterated format lists 14-41

iterative DO group
with index 11-10
without index 11-9

K indicator 3-41

KEY option
in DELETE statement 15-10
in READ statement 15-10, 15-15
in REWRITE statement 15-11

KEYED attribute 15-7

KEYED record input/output 15-8

KEYFROM option
in LOCATE statement 15-20
in WRITE statement 15-8

KEYTO option
in READ statement 15-15

keywords
in statements 5-14
versus names 5-3

LABEL
attribute 3-45

LABEL constant names 8-37

label prefix
for DO statement 11-9
for END statement 11-9
purpose of 5-13
to declare a name 6-8

language-defined conditions
references 13-6
13-2

layout conventions 11-20

LBOUND function 9-53

left-adjusted string 14-29

left-major order of arrays 3-57

LENGTH function 9-40

length of string 2-2, 3-16, 9-40

letters 5-2

level numbers 3-49

level references 8-16

level-one variable 3-49

lexemes
%INCLUDE macros 5-7
classification of 5-12
identifiers 5-3
isubs 5-7
literal constants 5-5
operators 5-6
pictures 5-7
punctuators 5-6
separators 5-10
5-2

LIKE attribute 6-11

limit of DO group 11-13

LINE format item 14-39

line number 14-5

LINE option
in PUT statement 14-12

line size 14-5

linemark 14-1

LINENO function 9-70

LIST option
in GET statement 14-11
in PUT statement 14-12

list-directed input/output 14-20

literal constants
as lexemes 5-5

LOCAL attribute 11-17

local transfer of control 11-17

LOCATE statement
syntax diagram A-16
15-20

location of variables 8-10

locator
attributes 3-45
values 2-4

locator qualifiers 8-22

locator-qualified variable reference
locator-qualified deletion 8-30

locator-qualified variable references
8-21

locator-qualifiers
deletion of 8-30

LOG function 9-22

LOG10 function 9-22

LOG2 function 9-22

logical operators 9-41

LOW function 9-47

macro names 5-7

major names 8-8

major types 10-2

major variable 3-49

management class 7-10

mantissa of FLOAT value 3-6, 3-7

manuals on PL/I 1-12

mathematical operations 9-19

MAX function 9-11

maximum length of string 3-16

member references 8-16

members of structures 3-49

MIN function 9-11

MOD function 9-14

mode attributes
for pictured storage 3-23
guidelines for choice of 3-14
3-5

multi-valued functions 9-20

multiplication operator 9-7

MULTIPLY function 9-10

NAME condition 14-46

name-sequence
for declaration 6-16
for name reference 6-17

names
correspondence with storage 7-14
deletion of 8-28
versus keywords 5-3

natural logarithms 9-21

nested expressions 8-2

nesting
of blocks and groups 5-17
of DO statements 11-16
of IF statements 11-6

next record indicator
initialization of 15-6
15-2, 15-4

no-suppression digit indicator 3-30

non-computational values 2-5

non-iterative DO group 11-6, 11-16

non-local transfer of control 11-17

non-numeric indicator 3-43

non-numeric PICTURE attributes 3-22

nonstandard operations 9-3

NONVARYING
  attribute 3-17

normalized bounds of an array 3-56

normalized structure levels 3-51

not operator 9-41

NULL function 9-50

null record 15-2

null statement
  in IF statement 11-8
  syntax diagram A-17

NULLO function 9-50

number system 3-6

number-of-digits in precision 3-7

numeric PICTURE attributes 3-22

occupation record 3-47

occurrence of conditions 13-12

OFFSET
  attribute 3-45
  function 9-65
  values 2-4

ON statement
  syntax diagram A-17
  13-11

ON units
  defaults 13-15
  establishing and reverting 13-10
  for file communication 13-19
  in general recursion 12-43
  restrictions 13-14
  13-14

ONCHAR
  function 9-76
  pseudo-variable 10-13

ONCODE function 9-74

ONFIELD function 9-75

ONFILE function 9-77

ONKEY function
  use of 15-21
  9-75

ONLOC function 9-74

ONSOURCE
  function 9-76
  pseudo-variable 10-13

OPEN statement
  for record files 15-6
  for stream files 14-8
  syntax diagram A-18

operands 8-53

operations
  conventions for
    mathematical operations 9-19
    string operations 9-30
    9-3
  conventions for arithmetic
    operations 9-4
  general rules 9-2
  kinds of
    address 9-48
    AREA 9-48
    arithmetic 9-3
    array 9-52
    conversion 9-55
    mathematical 9-19
    strings 9-30
    system variable 9-69
  nonstandard 9-3
  summary of 1-4
  9-1

operator expressions 8-52

operators
  as lexemes 5-6
  classification of 5-6

optimization of expression evaluation
  8-5

options
  general form 5-14

OPTIONS attribute 12-20, 12-24

or operator 9-41

order of expression evaluation 8-5

ordinary external regions 7-4

OUTPUT attribute
  for record files 15-7
  for streams 14-9

OVERFLOW condition
  for arithmetic target 4-8
  for conversion 4-21

overlapping string targets 10-5

P format item 14-34

padding for strings 3-17, 9-31

PAGE format item 14-39

PAGE option
  in PUT statement 14-12

page size 14-5

pagemark 14-1

PAGENO
  function 9-70
  pseudo-variable 10-12

PARAMETER variables
  attribute 7-10
  guidelines for 7-34
  7-20

parameters
  asterisk extent 12-7
  restrictions on 12-19
  12-19

parent designator 12-39, 12-40

parenthesized expressions 8-3

parenthesized iterated list 14-22

partial BASED variables 7-28

partially-qualified reference 6-18

passing arguments 12-2

permanent internal regions 7-4

phase of iterative DO 11-11

PICTURE
  attribute
    complex 3-42
    fixed-point 3-22, 3-26, 3-29
    floating-point 3-22, 3-27, 3-41
    non-numeric 3-22, 3-43
    numeric 3-22
    3-21
  classification of pictures 3-22
  guidelines for 3-44
  indicators
    arithmetic decimal point 3-39
    decimal-point 3-32
    digit 3-22
    dollar 3-34
    drifting-dollar 3-37
    drifting-sign 3-36
    floating-point 3-41
    insertion characters 3-38
    no-suppression digit 3-30
    non-numeric 3-43
    scale-factor
      fixed-point 3-40
      floating-point 3-42
    sign 3-33

PICTURE (cont)
  indicators
    zero-suppression 3-35
    3-21
  lexemes 5-7
  storage
    arithmetic assignments 3-27
    arithmetic fetches 3-28
    character-string assignments 3-25
    character-string fetches 3-26
    interpretation of 3-23
    3-20
  values
    arithmetic 3-26
    character strings 3-24
    2-3

picture format item
  character 14-37
  fixed-point 14-35
  floating-point 14-37
  use of 3-44
  14-34

PL/I Language Manual 1-14

POINTER
  attribute 3-45
  function
    standard 9-65
  in recursion 12-37
  values 2-4

POSITION attribute 7-10, 7-31

power operator 9-9

precedence for operators 8-54

PRECISION
  attribute 3-7
  guidelines for choice of 3-15

PRECISION function 9-63

prefix of statement
  condition prefix 13-8
  procedure statement 12-18
  5-13

prefix operators 8-53

prefix sign operators 9-6

PRINT attribute 14-9

priority for operators 8-54

PROCEDURE block
  as program structure 5-18
  external 5-20
  in general recursion 12-41
  sequential execution of 11-2

PROCEDURE statement
  prefix of 12-18
  sequential execution of 11-2
  syntax diagram A-19

PROCEDURE statement (cont)
12-17

procedures
activation 12-10
deactivation of 12-12
execution 12-11
exit from 12-11, 12-15
recursive 12-20, 12-29
12-1, 12-17

PROD function 9-54

product operator 9-7

program
guidelines for 5-20
layout conventions 11-20
structure of 1-3, 5-17
syntax 5-1
validity 1-9
5-20

program flow 11-1

programmed function references 8-42

programmer-defined conditions
references 13-7
13-6

promotion of aggregates 4-18

pseudo-variable
in DO statement 11-15

pseudo-variables
in input/output statements 14-23
interpretation of 10-9
list of 10-4

punctuators
as lexemes 5-6

PUT statement
syntax diagram A-20
14-12

R format item 14-13

READ statement
for KEYED input/output 15-10
for SEQUENTIAL input/output 15-12
syntax diagram A-21

REAL
attribute 3-5, 3-23
function 9-18, 9-58
pseudo-variable 10-9
values 2-1

real part 10-9, 3-5

RECORD condition 15-23

record input/output
based 15-14
closing files 15-6

record input/output (cont)
conditions for 15-20
data sets 15-1
files 15-3
keyed 15-8
opening files 15-6
operations 15-5
sequential 15-12
15-1

recursion
activation indexes in 12-37
activation variable reference 12-40
BEGIN block 12-42
chained 12-34
FORMAT statement 12-42
general 12-36
GOTO statement 12-44
ON units 12-43
parent designators 12-39, 12-40
pointers in 12-37
PROCEDURE block 12-41
statement address constant
reference 12-40
surface 12-30
with arguments 12-33
without arguments 12-31
12-29

RECURSIVE keyword 12-20

recursive procedures 12-20

REFER option 7-24

reference
built-in function reference 8-45
destination of GOTO
restriction 11-18
11-16
locator-qualified 8-21
programmed function reference 8-42
shortened references 8-27
simple 8-9
structure-qualified 8-16
subscripted 8-12
to variable 8-8

references
constant 8-36

related arithmetic types 3-26

related character types 3-24

relational operators
for address values 9-48
for arithmetic values 9-10
9-38

remote format item 14-13

remote format items 14-41

REPEAT control for DO 11-12

replicators
in picture 3-21

replicators (cont)
in INITIAL attribute 7-36
in string constants 8-34

resolution of names
rules for 6-18
6-15

RETURN statement
as procedure exit 12-11, 12-15
assumed 11-2
syntax diagram A-22
12-22

RETURNS attribute 12-19

REVERSE function 9-40

REVERT statement
syntax diagram A-22
13-12

reverting ON units 13-10

REWRITE statement
for KEYED input/output 15-11
syntax diagram A-22

right-hand-side expression 10-4

ROUND function 9-13

rounding
for conversion 4-8
for pictured strings 3-31

S indicator
drifting 3-36
leftmost 3-33

scalars
values 2-5

scale attributes
guidelines for choice of 3-15
3-5

scale-factor in precision 3-8

scale-factor indicator
fixed-point 3-40
floating-point 3-42

scope
guidelines for 7-14
of condition prefix 13-8

scope attributes 6-22

scope of block 5-18

SCOPES 7-13

SEARCH function 9-43

self-describing structures 15-18

separation rules 5-10

separator lexemes 5-10

SEQUENTIAL attribute 15-7

sequential execution 11-2

SET option
in LOCATE statement 15-20
in READ statement 15-15
7-22

side effects 8-6

SIGN function 9-17

sign indicator 3-33

sign operators 9-6

sign-manipulation functions 9-16

SIGNAL statement
syntax diagram A-23
13-13

signalling of condition 13-13

significant digits 3-8

simple BASED variables 7-26

simple declarations 6-4

simple DEFINE variables 7-30

simple variable references 8-9

SIN function 9-24

SIND function 9-25

single precision 3-7

single-value control for DO 11-11

SINH function 9-28

SIZE condition
for arithmetic target 4-8
for conversion 4-20
for pictured strings 3-31

SIZE function 9-72

SKIP format item 14-39

SKIP option
in GET statement 14-11

SNAP keyword 13-11

spaces
as lexemes 5-10
5-2

special array functions 9-54

special characters 5-2

special conversion functions 9-65

special string target 10-6

SQRT function 9-23

stack frame
in recursion 12-29
12-10

stacking CONTROLLED variables 7-18

statement
address values 2-3
classification of 5-16
parts of
attributes 5-15
body 5-14
clauses 5-14
keywords 5-14
options 5-14
prefix 5-13

statement address constant references
12-40

STATIC variables
attribute 7-10
guidelines for 7-33
7-17

statistical analysis functions 9-29

status indicator 14-5, 15-4

storage 3-1

storage class attributes
guidelines for 7-32
6-22

storage classes 7-15

STORAGE condition 7-40

storage conditions 13-4

storage management
conditions for 7-39
functions 9-71
fundamental principles 7-3
fundamental prinicples 7-3
operations 7-5
7-1

storage regions
diagrams 7-4

storage types
addresses 3-44
aggregates 3-49
areas 3-46
arithmetic 3-4
conversion of 4-1
of expressions 8-3
ordinary strings 3-16
pictured strings 3-20
3-2

storage unit
box 3-2

storage units
contents 3-2
data frame 3-10
designator 3-2
examples of 3-10, 3-18
interpretation of 3-11
storage type 3-3
3-1

STREAM attribute 14-8

stream input/output
closing files 14-7
conditions for 14-44
data sets 14-1
data-directed 14-14
edit-directed 14-25
files 14-4
list-directed 14-20
opening files 14-7
operations 14-6
statements 14-11
STRING option 14-43
14-1

stream pointer
initialization of 14-8
14-5

streams
input 14-3
output 14-3
pseudo 14-4

STRING
function 9-66
pseudo-variable 10-11

string format items 14-28

STRING option
in GET statement 14-12
14-43

STRING options
in PUT statement 14-12

string overlay BASED variables 7-27

string overlay DEFINED variables 7-31

string-type attributes 3-16

strings
assignment of 10-2
attributes 3-16, 3-21
constant literals 8-34
constants
as lexemes 5-5
implicit string targets 4-5
operations 9-30
ordinary 3-16
padding 3-17
pictured 3-20
values 2-2

STRINGSIZE condition
for conversion 4-22
for string target 4-10

structure declarations 6-5

structure-qualified variable reference
name deletion 8-28

structure-qualified variable
references 8-16

structures
storage 3-51
storage types 3-50
values 2-4
3-49

subscripted variable reference
subscript-list deletion 8-27

subscripted variable references 8-12

SUBSCRIPTRANGE condition
in GOTO statement 11-17
8-12

subscripts
deletion of 8-27
8-12

subsets of PL/I
for business programming 1-7
for scientific programming 1-5
for system programming 1-8

SUBSTR
function 9-33
pseudo-variable 10-10

SUBTRACT function 9-10

SUM function 9-54

surface recursion 12-30

switch GOTO statement 11-16

syntactic validity 5-1

syntax
general rules 5-1
of statements
notation A-1
A-1

syntax of statements
conventions A-6
parts statement A-5

SYSIN stream file 14-10

SYSPRINT stream file 14-10

system counter functions 9-70

system variable 13-15

system variable operations 9-69

SYSYEM keyword 13-11

TAN function 9-24

TAND function 9-25

TANH function 9-28

targets
for assignment 10-4
for conversion 4-2
implicit 4-4

termination conditions 13-5

test
in IF statement 11-4
in WHILE option 11-9, 11-11, 11-12,
11-13

texts on PL/I 1-12

THEN clause 11-3

TIME function 9-70

TITLE option
for record files 15-6
for streams 14-8

transfer of control 11-9

TRANSLATE function 9-44

transmission option
in GET statement 14-11
in PUT statement 14-12

TRANSMIT condition
for record input/output 15-23
for stream input/output 14-46

trigonometric functions 9-24

true value 2-3

TRUNC function 9-12

truncation functions 9-12

truncation of arithmetic values
for conversion 4-8

UNALIGNED attribute 3-62

unconnected aggregates
restriction of 12-3

undefined expression evaluation 8-6

UNDEFINEDFILE condition
for record input/output 15-23
for stream input/output 14-47
14-8

UNDERFLOW condition
for arithmetic target 4-8
for conversion 4-21

UNSPEC
function 9-67
pseudo-variable 10-12

UPDATE attribute
for record files 15-7

usage categories 7-12

V indicator 3-39

V. indicator 3-32

VALID function 9-68

validity of programs 1-9

values
address 2-3
aggregate 2-4
AREA 2-4
arithmetic 2-1
array 2-4
assignment of 10-1
classification of 2-5
conversion of 4-1
kinds of values 2-1
storage for 3-1
string 2-2
structure 2-4

variability attributes 3-17

variable names
attributes for 6-21

variable references
locator qualified 8-21
shortened references 8-27
simple 8-9
structure-qualified 8-16
subscripted 8-12
8-8

VARIABLE storage
attribute 7-10
7-13

variables 3-1

VARYING
attribute 3-17

VERIFY function 9-43

WHILE option
in FORTRAN control 11-13
in REPEAT control 11-12
in single-value control 11-11
without index 11-9

WRITE statement
for KEYED input/output 15-8
for SEQUENTIAL input/output 15-12

WRITE statement (cont)
syntax diagram A-23

X format item 14-38

X indicator 3-43

Y indicator 3-35

Z indicator 3-35

zero-suppression indicator 3-35

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

TITLE: SERIES 60(LEVEL 66)/6000 PL/I

ORDER NO.: DE05, REV. 0

DATED: MARCH 1976

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by appropriate technical personnel and action will be taken as required. If you require a written reply, check here and furnish complete mailing address below. ☐

FROM: NAME _____ **DATE** _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

CUT ALONG LINE

FOLD ALONG LINE

FIRST CLASS
PERMIT NO. 39531
WALTHAM, MA
02154

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

FOLD ALONG LINE

Honeywell

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

16008, 1676, Printed in U.S.A.

DE05, Rev. 0